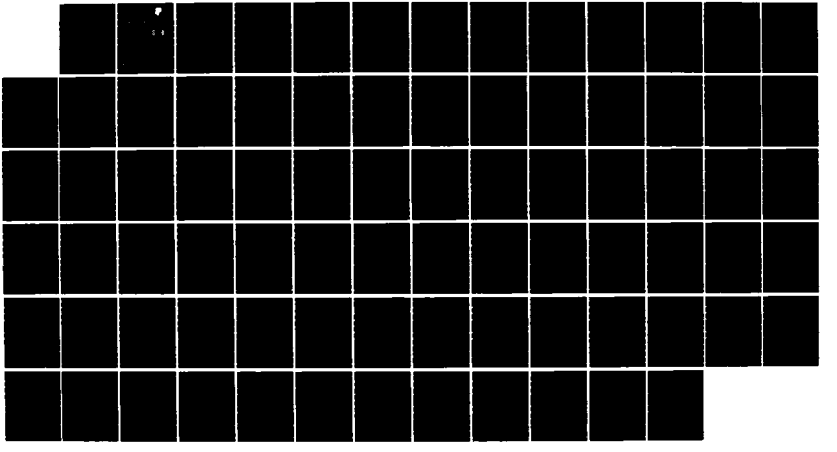
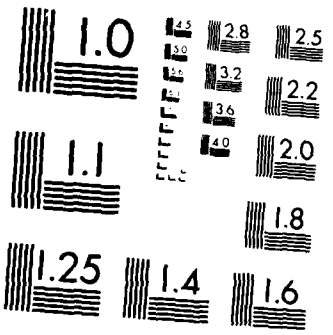
AD-A173 218 THE INA JO SPECIFICATION LANGUAGE: A CRITICAL STUDY(U) 1/1
MITRE CORP BEDFORD MA J D GUTTMAN JUL 86 MTR-9755
RADC-TR-86-47 F19628-84-C-0001

UNCLASSIFIED F/G 9/2 NL





(12)

RADC-TR-86-47
Final Technical Report
July 1986

THE INA JO SPECIFICATION LANGUAGE: A CRITICAL STUDY

The MITRE Corporation

Joshua D. Guttman

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MTR-9755	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-86-47
6a. NAME OF PERFORMING ORGANIZATION The MITRE Corporation 6b. OFFICE SYMBOL (If applicable) D-75	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTC)
6c. ADDRESS (City, State, and ZIP Code) Burlington Road Bedford MA 01731	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTC	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628-84-C-0001
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 64740F; PROJECT NO: 2239; TASK NO: PR; WORK UNIT ACCESSION NO: OJ
11. TITLE (Include Security Classification) THE INA JO SPECIFICATION LANGUAGE: A CRITICAL STUDY	
12. PERSONAL AUTHOR(S) Joshua D. Guttman	
13a. TYPE OF REPORT Final 13b. TIME COVERED FROM Oct 84 TO Sep 85	14. DATE OF REPORT (Year, Month, Day) July 1986 15. PAGE COUNT 84
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Formal Development Methodology (FDM)
09	02		design verification
12	01		KVM

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Ina Jo, a language for formal specification and verification, provides a lucid vehicle for the logical description of software systems under design. The present paper gives a self-contained critical account of the full range of features of the languagel Concrete examples illustrate its strengths and weaknesses, and an approach to giving a formal semantics for the language is sketched. The paper addresses issues fundamental to the design of specification languages, but is also suitable as a tutorial introduction to the Ina Jo language.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL John C. Faust	22b. TELEPHONE (Include Area Code) (315) 330-3241 22c. OFFICE SYMBOL RADC (COTC)

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1
	ABSTRACT MACHINES	2
	A SIMPLE EXAMPLE	3
	CORRECTNESS OF A SPECIFICATION	4
	A NONPROCEDURAL LANGUAGE	6
	THE PARTS OF A SPECIFICATION	8
2	THE STATIC PART OF AN INA JO SPECIFICATION	9
	TYPES	9
	CONSTANTS AND AXIOMS	13
	STATIC SEMANTICS OF INA JO	15
3	THE DYNAMIC PART OF AN INA JO SPECIFICATION	19
	VARIABLES AND INITIAL CONDITIONS	19
	TRANSFORMS	20
	NO CHANGE	23
	AN EXAMPLE	27
	DYNAMIC SEMANTICS	31
4	EXPRESSING REQUIREMENTS IN INA JO	34
	CORRECTNESS CONDITIONS	34
	CORRECTNESS AND SECURITY	42
	LIMITS TO THE FDM MODEL OF CORRECTNESS	54
5	MULTI-LEVEL SPECIFICATIONS	57
	INTRODUCTORY	57
	CONTINUING OUR EXAMPLE: MAPPING TYPES, CONSTANTS AND VARIABLES	58
	RULES FOR MAPPING TYPES, CONSTANTS, AND VARIABLES	61
	CONTINUING OUR EXAMPLE: MAPPING TRANSFORMS	63
	THE MEANING OF MAPPINGS	67
	THE FIDELITY OF REFINEMENTS	70
	CONCLUSIONS	73
	DISTRIBUTION LIST	74

SECTION 1

INTRODUCTION

FDM, the Formal Development Methodology, was developed by the System Development Company beginning in the mid-seventies. One of three specification and verification systems endorsed by the National Computer Security Center, it has been used for design verification in a number of secure computing systems projects. Most notable among these is probably KVM, carried out at SDC, which attempted to transform IBM's Virtual Machine operating system into a kernelized secure system.

FDM aids in the design of large software systems. It allows the user to describe overall requirements for the system he is developing, and provides a structured environment to help him write the system specifications in a step by step way. As the system specifications evolve, the FDM tools help the user prove that the requirements will hold for any system satisfying the current version.

It is important to note at the start what FDM is and is not. The methodology helps us to specify systems, and to verify that the specifications fit our requirements. It does not help with implementation. A human being must do the final coding, into some programming language, entirely by hand. Moreover, although FDM supports proofs that the specifications satisfy system requirements, FDM does not currently provide any machinery to help prove that a collection of programs is a correct implementation of the specifications. There have long been plans to add this capability to FDM, but they appear to have run aground. We will return to this issue in a subsequent report, in the meantime emphasizing only that FDM allows us to produce verified specifications but not, at least for the present, verified code.

The specifications and system requirements are expressed in the language Ina Jo, which was created just for FDM. Ina Jo is not a programming language, as it is not used to represent algorithms. Rather, it is a descriptive language: it is used to describe the results of procedures in a way that does not depend on the algorithms chosen to implement them.

Descriptive languages are a natural tool for writing specifications. Specifications formulate the overall goals of a software package, and can thus guide the choice of algorithms and data structures for the software. Therefore, we want to express

specifications in a language that does not commit us to particular algorithms and data representations.

ABSTRACT MACHINES

An Ina Jo specification describes the system under development as an abstract machine. The notion of an abstract machine gives a flexible framework useful in a variety of areas of computer science [Dijkstra, 1972, Tannenbaum, 1984].

At the austere end of the spectrum of abstract machines we have Turing machines. Near the luxuriant extreme, we have, for instance, the UNIX shell. The user of a UNIX(TM) shell has a variety of commands available; by typing in one of these commands, he makes the machine perform an operation. Exactly what the machine will do is not determined simply by what appears on the command line. It depends also on the internal environment, for instance, on the current working directory, on the contents of files, and so on. The effect of a command, in turn, is to change the internal environment of the machine, and possibly also to cause some output.

An abstract machine has an internal environment, or state, as we shall call it, and a collection of commands. The commands cause the machine to pass from one internal state to another. We will often also call the commands operations. Just as a command may have arguments supplied by the user, we will allow operations to have user-supplied parameters.

The machine has a particular initial state, or a set of possible initial states, that every session starts off from. The "user" of the abstract machine begins by choosing an operation, and possibly also some arguments to the operation. The machine reacts by shifting from its initial state to a new state. As the session continues, the user repeatedly chooses commands, causing the machine to shift from state.

In Ina Jo, each machine comes equipped with a particular list of variables; the current values of these variables make up the state of the machine. That is, an Ina Jo machine-state consists of the present values of this list of variables. Ina Jo operations cause their values to change. Thus, an Ina Jo specification must describe a machine by saying what its variables are, what kinds of values each one of them can assume, and how the various operations change the variables.

A SIMPLE EXAMPLE

As an example, let us describe a simple desk calculator for integer arithmetic. Consider what follows as a highly informal specification for an abstract machine. Later we can rewrite it as a formal Ina Jo specification. The machine will maintain a stack: each new number input will be placed on top of the stack. Each arithmetic operation -- say, addition, subtraction, multiplication, and division, -- will cause the top two numbers to be removed from the stack and combined. The result will be placed on top of the stack.

To calculate the value of an expression using this kind of calculator, one enters the expression in reverse Polish notation. Thus for instance, putting (3 + 5) * 7, into RPN, we get

3 5 + 7 *.

We now enter the integers 3 and 5, apply the add operator, enter 7, and apply the multiplication operator. When we apply the addition operator, the numbers 5 and 3 are removed from the stack and added; 8, their sum, is placed on the stack. When we apply the multiplication operator, the numbers 7 and 8 are removed and multiplied. Their product, 56, is placed back on the stack.

The desk calculator will have two variables. One is the stack itself; the other will be a display. The value of the stack can be any list of integers. The value of the display can be any integer. The idea is that the display will contain the number on top of the stack, and that it will be implemented in some hardware display device rather than in memory.

The operations of the machine include the four arithmetic operations. These do not take user-supplied arguments, but simply combine the top two operands on the stack. There must also be a "push" operation, which takes an integer argument. The push operation changes the values of both the variables. It sets the display variable to the value of its argument, and updates the stack variable by adding its argument to the top of the previous stack value. It is also convenient to have a "pop" operation, like the "clear error" key on a calculator, which discards the top value on the stack. It updates the display to show the next value on the stack.

Thus, we are considering a machine with two variables, one taking integer values, the other taking lists of integers as values. The machine has six operations. One of them, namely "push", takes an integer as an argument.

How shall we choose the initial state of our machine? Clearly, we want the stack to start off null, but it doesn't matter what the display shows to begin with. After all, every time we use the machine we will begin by pushing an integer onto the stack; the value we push will replace the value in the display, which can thus have no role in the future life of the machine. We could choose an initial value for the display arbitrarily, say, zero. Or else the specification could omit any mention of the initial value of the display, leaving that as an "implementation detail" to be resolved at the time the machine is actually coded. The second idea seems more sensible: why should we specify a fact which makes no difference? The second approach is definitely cleaner, as the specification will be correct no matter what the initial display is. A basic principle behind the FDM approach is that nothing should appear in the specification unless it is necessary for the correctness of the specification.

CORRECTNESS OF A SPECIFICATION

What does it mean to say that a specification is correct? Presumably, that any machine built to the specifications will compute the right value for any expression we enter into it. There are two parts to this: not only must the right value end up on the top of the stack, but the display must also show it. Thus to show that the specification is correct, we will want to show that the right numbers end up on the stack, and also that the display always shows the value on the top of the stack when the stack contains values (when it is not null).

To prove that the right numbers appear on the stack, we want to show that each of the operations causes the stack to be updated in the right way: "push" by pushing its argument to the old stack, "add" by adding the top two numbers, etc. This is a fact about the relation between one state of the stack and the next state of the stack. That is, whether a state is correct depends on what the preceding state was. As for the display, to prove that it contains the correct value, we must show that it equals the value on top of the stack whenever there is at least one value on the stack. This is not a fact about the relation of one state of the variables to the next. We do not have to know what the preceding values of the variables were to know that the value in the display is the same as the value on top of the stack. The current values are all we need to know.

Examples like this suggest that we should separate two slightly different kinds of correctness condition. The first describes a connection between one state and the next. The requirement that our

calculator has the right numbers on its stack is of this sort. The other kind of correctness condition says something about each state the abstract machine will ever enter. Our second requirement was of this kind; it laid down that our calculator should never enter a state in which the stack contains at least one number and the display does not show the top number on the stack.

FDM does separate these two kinds of correctness conditions. SDC terminology speaks of "constraints" and "criteria" respectively. A constraint requires that, no matter what operation we may choose to apply to the current state, the current state will have the right relation to the next state of the machine. A criterion is a statement about every state that the machine can ever enter. It requires that, starting from an initial state and applying whatever operations we choose, the machine should never enter the wrong kind of state.

A constraint describes the succession of states, the relation of one to the next. A criterion expresses a property of individual states, and demands that the machine never work itself into a state which does not have the property.

Let us now take a look at the Ina Jo language, to see how operations and correctness conditions are actually expressed in the FDM approach. In our exposition we will follow a slightly unusual strategy, as we will pretend, throughout the next section, that an FDM specification is only a single-layered affair.

In fact this is not true: FDM allows us to write the complete specification in several stages, or levels. The successive layers can use quite different vocabularies, and can be based on quite different data structures. They are connected by logical predicates called "mappings". The ability to write layered specifications is a valuable feature of FDM, though perhaps not as valuable as it has been made out.

I will not be discussing the layering in FDM during the first part of my exposition, for two reasons. First, it complicates the syntax and especially the semantics of Ina Jo. It is not so difficult to sketch the outlines of a rigorous approach to the language, if one restricts oneself to a single level. But I think that the full, multi-level version is a great deal harder to explain. Moreover, the natural way of giving the semantics is to convert the multi-level structure back into a single level description, and interpret the latter. Corresponding to the more complicated semantics, the idea of correctness is essentially more complicated for multi-level specifications. We will see that much more has to be proved to show that a specification satisfies its requirements if it is multi-level.

Second, the idea behind the levels in FDM is not what one first expects. Indeed, several different processes are involved as one goes from one level to the next. I think it is easier to understand what is going on, and why FDM levels are connected in the way they are, if one already has a clear understanding of the individual levels.

With this warning in mind, let us proceed to talk about an Ina Jo spec as if it consisted of a single level. In Chapter V we will introduce the multi-level apparatus of Ina Jo.

A NONPROCEDURAL LANGUAGE

The Ina Jo language, as I mentioned above, is not a programming language. By that I mean that it is not procedural -- it does not express procedures or algorithms for carrying out computations. Rather, it is a descriptive language, designed to express what a particular operation does -- the effect it has -- rather than how the operation achieves its effect.

Because Ina Jo is descriptive rather than procedural, it has a syntax rather different from programming languages. It has no looping constructs, no assignment statements, no procedure calls. Instead, it is modeled -- perhaps loosely -- on predicate logic.

An Ina Jo specification for a particular operation contains a predicate describing the effect of the operation. This predicate describes the new values of the state variables in terms of their old values and the parameters to the operation. For instance, for the "push" operation in our desk calculator, we would have to state that the new value of the Stack variable is equal to the result of prefixing the parameter to the old value of the Stack variable. We also want to say that the new value of the Display variable is equal to the parameter. The "add" operation would be described by saying that the new value of the Stack variable is equal to:

```
(cons (+ (car Oldstack)
         (cadr Oldstack))
      (cddr Oldstack))
```

as one would write it in LISP notation.

To rewrite this expression in Ina Jo we need three operators. One is ";". It means to prepend -- l;a is the result of putting a at the front of the list l. The second, ":", means tail.

Oldstack:3, for instance, means the part of oldstack beginning with the third element. Finally, l.n denotes the nth element in the list l.

Using these operators, we can rewrite the expression as

```
(Oldstack:3);(Oldstack.1 + Oldstack.2) .
```

The new value of the display should be the top value on the new stack.

We need, then, some convention letting us refer to the values of a variable before and after an operation is applied. The Ina Jo convention is that a variable name refers to the value the variable had before the operation was performed. To refer to the new value of the variable, Ina Jo uses the "new value" operator N". Thus we could write the predicate describing the effects of the "Add" operation as

```
      N"Stack = (Stack:3);(Stack.1 + Stack.2)
  &   N"Display =  (N"Stack).1 .
```

It is important to keep clearly in mind that this is a predicate, not a command. The sign "=" means equality, not assignment. Thus, for instance, the order of the two conjuncts is irrelevant. The formula means just the same as

```
      N"Display =  (N"Stack).1
  &   N"Stack = (Stack:3);(Stack.1 + Stack.2),
```

although the latter is harder for a human being to read. Similarly, we could write it in any logically equivalent form, such as

```
      ~(N"Display =  (N"Stack).1
       =>
       ~(N"Stack = (Stack:3);(Stack.1 + Stack.2) ) ),
```

which would be even more unreadable. (Note that Ina Jo uses the familiar symbols "~", "=>" and "&" for "not", "implies", and "and".)

Similarly, assuming we call the argument to the "push" operation "arg", we can describe the effects of the operation by the predicate

```
      N"Stack = Stack;arg
  &   N"Display = arg .
```

This says that the new value of Stack is the result of prepending arg to the old value, and that arg is the new value for Display.

These predicates have a somewhat unusual characteristic. Namely, they are deterministic, in the sense that there is only one possible new value for the variables, given their old values and, with "push", the parameter. Ina Jo does not require the effect of an operation to be deterministic, and often one wants to use nondeterministic predicates, especially in high-level specifications.

Let us extend the specification for our desk calculator somewhat to introduce a nondeterministic predicate. Suppose we want, possibly for bookkeeping purposes, to keep track of how much computing has been done. Moreover, the various operations may require different amounts of computing time -- multiplication requiring more than addition and subtraction, and division requiring still more. However, we probably do not want to write the exact amounts into the top level specification. That is an implementation level detail that will depend on the hardware and so forth. We will add a new variable TimeUsed, and add the clause

N"TimeUsed > TimeUsed

to the effects of each of the operations. Here we have to use a non-deterministic predicate, because we do not yet know how large the increases should be for the different operations.

THE PARTS OF A SPECIFICATION

The items contained in a (single-level) Ina Jo specification fall into three essentially different parts, which we will call the static, dynamic and correctness parts. The correctness part contains the criterion and constraint for the specification; we will discuss it last. The dynamic part consists of the declarations of state variables, together with the descriptions of the operations of the abstract machine. I call it the dynamic part because it is concerned with what the machine does. The dynamic part, however, presupposes another kind of information, contained at the beginning of the specification. This information, contained in the static part, describes the types of values that the state variables can take on, and furnishes a vocabulary for talking about those values and their relationships.

The next three chapters will be devoted to these three parts in turn.

SECTION 2

THE STATIC PART OF AN INA JO SPECIFICATION

The static part of a specification contains types and "constants". An Ina Jo constant is an object or function which is independent of the state of the abstract machine. It either gives a name to a member of one of the types, or gives us the vocabulary to describe the relations between objects of one or more types. We are able to include axioms which help to characterize the types and constants. This facility allows us to give an algebraic or logical description of the types and constants. We can think of the types, constants, and axioms as describing an underlying structure, or mathematical universe.

In the dynamic part of the specification, the abstract machine will be specified in terms of the objects in this structure and the constants defined on it. Its state variables always have values in the structure. The effects of the operations of the machine on the state variables will have to be specified in terms of the individual constants and function constants defined in the structure. Thus, the static part is the basis in terms of which the rest of the specification is written.

TYPES

Most modern programming languages divide objects into types, allowing only objects of a single type to be assigned to a variable. Ina Jo uses a similar approach. All data objects are divided into types, and each variable is associated with a particular type of object. Thus, each state variable is declared with a particular type, and its value must always be a data object of that type. Each parameter of an operation must also have its type, and it would be nonsense to try to perform an operation with an actual parameter belonging to the wrong type.

Ina Jo gives the user great flexibility in defining types, and also, as we shall see, flexibility in **not** defining types. For in an Ina Jo specification, not all types have to be defined in terms of basic, built-in types. Some types may be **unspecified**. These types will be defined only later, either in a more detailed level of the specification, or in the implementation. Unspecified types are important in top-down development, as one often wants to work out the operations one will need to apply to a set of data-objects before figuring out how to structure them.

Ina Jo has only two predefined types, namely boolean and integer. It would also be convenient to have a predefined character type. However, since it is easy to introduce user-defined types in Ina Jo, this is hardly a serious problem.

User defined types may be of several kinds. First, there are the unspecified types. These are declared in the simplest possible way. Namely, one places the identifier naming the type among the type declarations. For instance,

```
Type SecurityLevel,
     DatabaseObject,
     User
```

The effect of a declaration for an unspecified type is simply to cause the Ina Jo language processor to recognize the identifier as a type name. The user can then introduce variables on the type, define operations on it, and make assertions about the variables and operations. He can decide how to implement the type at a later time, basing his decision on this information.

Next come the enumerated types. Ina Jo declares an enumerated type by listing the elements -- at least two -- within parentheses; for instance,

```
Type Day = (sun, mon, tue, wed, thurs, fri, sat)
```

declares the type Day to have those seven elements. An enumerated type has an ordering: one element is less than another if it occurs earlier in the declaration. So in Day, mon < wed and wed < sat.

User defined types can also be constructed from given types. The language offers a large number of operators for constructing types, including taking sets, lists, and tuples. Thus the following are all valid declarations:

```
Type Unavailable = Set Of Days,
     WorkDay = Day >< integer,
     TimeCard = List Of WorkDay
```

An object of type Unavailable is a set of days, presumably representing the set of days on which someone is unavailable. An object of type WorkDay is a pair -- a tuple containing two items -- consisting of a Day and a number, designed to represent the number of hours worked in the Day. The notation "><" is supposed to look like a multiplication sign. It forms the Cartesian product of two types. One can build up tuples containing any number of items.

Ina Jo also allows the user to define subtypes. A subtype is a type all of whose members belong to some other type (called the supertype). A type which is not a subtype of any other type is called an ultra-type. Ultra-types are disjoint: no object belongs to two ultra-types.

A subtype may be unspecified, in the sense that the type declaration does not say which members of the supertype will belong to the subtype. Ina Jo uses the less-than sign to introduce an unspecified subtype:

```
Type User,
     SecurityLevel,
     DataBaseObject,
     PrivilegedUser < User,
     LowLevelObject < DataBaseObject
```

Subtypes can also be specified, meaning that the declaration determines which objects belong to the subtypes. If the supertype is an enumerated type, then the subtype is expressed in the same form.

```
Type Day = (sun, mon, tue, wed, thurs, fri, sat),
     WeekDay = (mon, tue, wed, thurs, fri)
```

If the supertype is not enumerated, one uses the (very powerful) operator T". This operator should be read "the type of". For instance,

```
Type MonthDay = T"i : integer (1 <= i & i <= 31)
```

declares MonthDay to be the type of integers between 1 and 31, the numbers which represent a day of a month. One might use this in

```
Type MonthDay = T"i : integer (1 <= i & i <= 31),
     Month = (jan, feb, mar, apr, may, jun, jul, aug,
                  sep, oct, nov, dec),
     Year = integer,
     Date = MonthDay >< Month >< Year.
```

Year is a synonym for integer. The two types have the same elements, or rather, the two identifiers "Year" and "integer" both refer to the same type. Continuing the same example, we could have

```
Person,                          /* Unspecified */
Time = T"i : integer (0 <= i & i < 24)
   ><  T"i : integer (0 <= i & i < 60),
Appointment = Person >< Time,
Calendar = Set of (Date >< Appointment).
```

This sequence of declarations defines a calendar as a data structure associating zero or more Appointments with each Date, where an Appointment associates a Person with a Time. This example ought to give the flavour of Ina Jo data types, which can be abstract, lying far from any particular implementation.

Formally, one can describe the T" operator as follows. Suppose t is a type and P(x) is a predicate of objects of that type. Then T"x:t (P(x)) is the type consisting of those objects of type t which satisfy P.

The operator T" is very powerful. In fact, it is too powerful, for two reasons. First, it is possible using T" to define "null" types, types no object can belong to, for instance

```
Type Nullity = T" i : integer (~( i = i ) ).
```

This is problematic, because the FDM Interactive Theorem Prover uses a logic which assumes implicitly that every type contains at least one object. Thus if a specification contains a type like Nullity, contradictions will arise, and it will be possible to use the ITP to prove that the specification satisfies any system requirement whatsoever. The papers [Korelsky and Sutherland, 1984] and [Platek, 1985], which originally pointed out this flaw, illustrate the problem by "proving" that the Bell-LaPadula security policy is satisfied by a specification which gives every user access to every file. They succeed because the specification contains one type which, although this is difficult to see at first, cannot contain any objects. Thus, the T" operator should be used only when one can be certain that the resulting type contains at least one object.

Unfortunately this problem is exacerbated by a second problem. A type declaration using T" can contain a complicated predicate. In fact, since Ina Jo contains the logical quantifiers **all** and **some**, it is possible to write predicates such that no algorithm can possibly decide whether the predicate is true of a given object. In cases like these it is hard to know what the type being defined really means; moreover, it will be impossible to implement the specification on a computer, which is of course an algorithmic machine. In these cases, and also in less extreme ones, it will be difficult to tell whether the type being declared is null or not. For instance, we could define the type Fermat to be the type of all quadruples (a, b, c, n) such that

```
(n > 2)  &  (a expt n = b expt n + c expt n)
```

In the seventeenth century Fermat claimed that this type is null, but mathematicians have been unable to prove his conjecture. There are, then, some uses of the T" operator for which one cannot tell whether the resulting type is null or not.

Since it can be hard to tell whether a type definition involving the T" operator is legitimate, it should be used with real caution. Even if there is no threat of contradiction, and if all type definitions are algorithmically decidable, reckless use of complicated types can lead to difficulties in implementation. For instance, they disguise the algorithmic complexity of programs. A program may look simple -- perhaps it just lists the elements of the type in order -- and nevertheless be slow, for instance with the type of all prime numbers up to some large bound. Similarly, programs with runtime type checking will consume a great deal of time when complicated types are present. One can question whether it was wise to introduce the operator into the language. It certainly seems like good practice to use the T" operator only with simple predicates.

CONSTANTS AND AXIOMS

Types would be useless if one couldn't declare constants and variables. In Ina Jo, the variables belong to the abstract machine one is specifying. That is to say, the variables are state variables of the abstract machine. Jointly, they determine the state of the machine; the purpose of the operations is to change their values. Constants are data structures whose value cannot be changed by any operation of the machine. They do not really belong to the abstract machine at all. Rather, they help to characterize the underlying structure, giving us a vocabulary for discussing the elements of the universe. We use this vocabulary in specifying the abstract machine and also in writing the correctness conditions.

Curiously, constants and variables do not have to belong to declared types. Rather, they can either be in a declared type, or else they can denote functions on the declared types. For instance, in a specification for an operating system secure in the sense of Bell-LaPadula, one needs a partial ordering of the security levels. Intuitively, a subject is allowed to read a file only if his security level dominates the level of the file, while he is allowed to write the file only if the level of the file dominates his level. These conditions are designed to prevent information from "flowing" from a high level file to a low level file. Moreover, the relationship between levels cannot be altered by any operation of the machine: a user cannot temporarily make Secret a lower classification than Confidential. The partial ordering is constant

in the sense that the operations of the abstract machine will not alter it. Therefore we can declare the partial ordering in the form:

```
Type SecurityLevel

Constant
  lteq(SecurityLevel, SecurityLevel):Boolean.
```

The expression **lteq** is a constant in that the operations do not alter it. But it is a function in that it "returns a value", one of the truth values True and False. Ina Jo terminology calls it a "constant function". This is surely an unfortunate expression, since most of the world uses "constant function" to mean a function which yields the same value for every argument, like the function (Lambda x. 0) which yields the value zero for every argument x. It would have been better to use the phrase "function constant" instead.

Usually, one does not know the exact value of a function at specification time: lteq would not be known to the specifier, who need not even know what security levels will be on the machine. This is one reason constants are important in a specification language. One must specify the fact that users cannot change the ordering of the security levels, so lteq must not be a variable. Yet one does not know just what the value will be, so one cannot write it out in full.

Even though one does not know the value of lteq at specification time, one does know something very important about it. Namely, one knows that it is a partial ordering. One must be able to specify that a constant will satisfy certain **axioms**.

An axiom in Ina Jo is an expression of Boolean type which one wants to require to be true in every implementation of the specification. It should contain only constants, together with predefined expressions of Ina Jo. It puts a constraint on the way that the constants will be implemented, because no implementation will be acceptable unless it makes the axioms true.

In the case of lteq, we want to require that it is a weak partial ordering. A weak partial ordering is a relation which is reflexive, transitive, and asymmetric. In Ina Jo, we can express these requirements by saying, respectively,

```
A"s:SecurityLevel (lteq(s,s))

A"s1,s2,s3:SecurityLevel (
```

```
        (lteq(s1,s2) & lteq(s2,s3))
        =>
         lteq(s1,s3)  )

    A"s1,s2:SecurityLevel (
        (lteq(s1,s2) & lteq(s2,s1))
        =>
         s1 = s2  )
```

The symbol A" is Ina Jo's universal quantifier; the phrase **A"s1,s2:SecurityLevel**, for instance, should be read "For all SecurityLevels s1 and s2, ... ".

These axioms involve just the one function constant "lteq", but an axiom may also involve several different constants. For instance, if Ina Jo did not contain the predefined integer functions of addition and multiplication, we would want to introduce them as function constants:

```
Constant Plus(integer, integer):integer
         Times(integer, integer):integer

Axiom    A"n1,n2,m:integer (
          Times(m, Plus(n1,n2))
         =
          Plus( Times(m,n1), Times(m,n2) ) )
       & etc.
```

The axiom describes a familiar fact about the way multiplication and addition link up, namely the distributive law.

STATIC SEMANTICS OF INA JO

The ideas we need in order to give a semantical account of the static part of Ina Jo are relatively simple. We interpret the static part of a specification by giving a many-sorted structure. The objects of the various types in the specification are associated with entities of appropriate sorts in the structure. Individual constants and function constants are interpreted by entities and functions defined in the structure. The most important restriction is that the interpretation must make the axioms true.

We will map each ultra-type to a sort. If S is a subtype of T, then we will interpret S by a subset of the sort T is mapped to. If a type is constructed from one or more base types, say by taking tuples or lists, then its sort should be consist of tuples or lists chosen from the base sorts. The predefined types integer and

boolean should always be interpreted by the set of integers and the set of truth values respectively, as that is what they are supposed to mean. As we expect each object in the structure to correspond to a specified object, an enumerated type should be interpreted by a set containing just as many objects as the type has; each constant of the enumerated type should be interpreted by a different member of the set.

The interpretations of individual constants and function constants will be constrained by the interpretations of the types. Suppose that type1, type2 and type3 are interpreted by the sets a, b, and c respectively. Then a constant declared

```
sam(type1, type2) : type3
```

will have to be interpreted by a two-place function defined for arguments from sets a and b. The function should yield values in c.

We can illustrate this way of interpreting the static part of Ina Jo with an extension of a previous example. Let us declare:

```
Type

        MonthDay = T"i : integer (1 <= i & i <= 31),
        Month = (jan, feb, mar, apr, may, jun, jul, aug,
                 sep, oct, nov, dec),
        Year = integer,
        Date = MonthDay >< Month >< Year

Constant

        Real(Date) : Boolean,  /* Really no Feb 30 */
        Next(Date) : Date,
        Prev(Date) : Date,
        YearLater(Date) : Date

Axiom
        A"d : Date
           ( Real(d)
           =>
             Real(Next(d))
           & Real(Prev(d))
           & Real(YearLater(d))
           & Next(Prev(d)) = d
           & Prev(Next(d)) = d
           )
```

How would we want to interpret this fragment of a specification?

Year must be interpreted by the set of integers, and MonthDay will be interpreted by the set of integers from 1 to 31. Month is an enumerated type. We must interpret it by a set containing exactly twelve elements. Let us choose the twelve months themselves. We will interpret the constant "jan" by the month of January; the constant "feb" by the month of February; and so on. A Date will be interpreted by a triple consisting of an integer from 1 to 31, a month, and an integer. Let us call these triples "date-triples". For convenience we will write them in the form <18 February 1985>.

The function interpreting Real must yield a truth value when applied to a date-triple. We will make it take the value true except for <31 April z>, <31 June z>, etc, and <29 February 4*z+1>, etc. We will interpret Real by the function R such that R(<x y z>) = True if

```
        y = January
    or  y = February & x < 28
    or  y = February & z mod 4 = 0
            & x = 29
    or  y = March
    or  y = April & x < 31
    or  ...

    or y = December;
```

otherwise, R(<x y z>) = False. If R is true of a date triple, we will call it a real triple.

The function YearLater is not too hard to interpret. We should map YearLater(<x y z>) to <x y z+1> unless x is 29, y is February, and z is divisible by 4. In that case the axiom causes trouble, as <x y z+1> will not be a real triple. So let us take <1 March z+1> instead.

Rather than giving the interpretation of Next and Prev directly, let us define an ordering on the real triples. If <x1 y1 z1> and <x2 y2 z2> are real triples, the we will say that <x1 y1 z1> is less than <x2 y2 z2> if:

```
      z1 is less than z2
  or z1 = z2
       and (y1 comes before y2 in the year
            or y1 = y2 and x1 is less than x2)
```

Each triple has an immediate predecessor and an immediate successor in this ordering. So we can interpret Prev by the function which maps each real triple to its immediate predecessor and -- for

completeness sake -- maps each unreal triple to itself. The function mapping each real triple to its predecessor, leaving unreal triples unchanged, will serve to interpret Next. This way, Next and Prev are inverses, so that the axiom will be true.

This is a concrete example showing how to interpret the static part of an Ina Jo specification. We chose a structure having two basic sorts, the integers and the months. MonthDays were interpreted by a subset of the integers; Dates by a sort consisting of triples. We interpreted the function constants by functions defined in the structure, in a way which makes the axioms true.

Because we have said how to interpret the basic notions, we have indirectly fixed the meanings of all of the compound expressions we can build up using operators belonging to Ina Jo. For the most part, the individual operators make straightforward contributions to building up the meanings of expressions in which they occur. The important point here is that the static semantics tells us a meaning for expression we can write using the Ina Jo language and the constants we have declared in our specification. A boolean expression, for instance, is an assertion which is either true or false in our structure. And a boolean expression with free variables is either true or false of a list of objects in the structure, assuming the objects are of the appropriate sorts.

Naturally, we have not given a thorough and rigorous semantics for the whole static part of Ina Jo. But it seems clear how to develop such an account on the basis of these ideas. Moreover, this sketch will help us clarify the semantics of the dynamic part, as well as the notion of correctness and the Ina Jo inter-level mappings.

SECTION 3

THE DYNAMIC PART OF AN INA JO SPECIFICATION

VARIABLES AND INITIAL CONDITIONS

Variables, like constants, may be functions on the defined types. For instance, if we want to keep track of who has accessed which files when, we could keep a variable Accessed:

```
Type User,
     File,
     Date

Variable
     Accessed(User, File, Date) : Boolean
```

This variable is in effect a function, which is intended to supply the value True if the user accessed the file on that date. Note, however, that there is a lot of free choice as to how to implement an Ina Jo variable. Accessed could be kept as an array. Or else we could associate a linked list with each user, containing the names of files and the dates on which they were accessed. The reader will be able to add quite a few alternatives. This is a basic feature of the Ina Jo approach. We should be able to write the specification without worrying about implementation details, such as how the variable Accessed should be represented. All the specifier needs to say is that he wants the system to be able to yield True or False when the question whether a user accessed a file on a particular date is asked. From the specifier's point of view, all that matters is that Accessed should behave like a logical function, yielding a value for each case to which it is applied.

An Ina Jo specification may contain axioms constraining the values of constants. Variables, by contrast, do not have axioms. Rather, the specifier can only supply conditions restricting the **initial** values of variables. The values they take on later will depend on what operations are applied. If the specifier wants to make sure that the values of a variable will always satisfy some requirement, he will have to prove it. To do so he must prove that the initial value of the variable will satisfy the requirement, and also that no operation of the abstract machine will take an acceptable value and transform it into an unacceptable value.

Although we will return to the details of these proofs below, an example is probably in order. Suppose that we are specifying a

system which is to obey the Bell-LaPadula model of security. We want to ensure that when a user reads a file, his security level dominates its level, and that when he writes to a file, the level of the file dominates his file. ("No read up, no write down.") We can think of this as a property which must always hold true of a variable Accessed. To make sure that it will always hold, we must ensure that no operation of the state machine allows an access violating the principle. This is sufficient to make sure that it will always hold true, assuming it held true at the beginning, in the machine's initial state. Therefore, we should stipulate it as an initial condition on the variable Accessed.

```
Type User,
     File,
     Date,
     SecurityLevel,
     AccessType = (Read, Write)

Constant
  lteq(SecurityLevel, SecurityLevel) : Boolean

Axiom /** As above -- lteq is a partial ordering **/

Variable
     Accessed(User, File, Date, AccessType) : Boolean,
     FileLevel(File) : SecurityLevel,
     UserLevel(User) : SecurityLevel,

Initial
     A"u:User, f:File, d:Date (
          (Accessed(u, f, d, Read)
               => lteq(FileLevel(f), UserLevel(u) ) )
        &
          (Accessed(u, f, d, Write)
               => lteq(UserLevel(u), FileLevel(f) ) )  )
```

We will discuss later how FDM allows us to prove that a property like this one remains true as the machine moves from state to state.

TRANSFORMS

There is one other kind of Ina Jo unit in addition to types, constants, and variables, namely transforms, which describe operations. These are really the most important of all: they describe what the abstract machine we are specifying will do. The FDM terminology for transforms is a little sloppy, and it is not

always clear just how SDC is using its words, so I shall adopt the convention that a "transform" is a patch of text written in Ina Jo, while an "operation" is what a transform describes: a state-changing procedure a particular machine can apply.

This distinction may seem a bit metaphysical, but it is in fact important to keep in mind. Specifications are always -- naturally -- incomplete in the sense that there are many different ways of implementing them. The role of the specification is simply to pick out a class of possible implementations, namely those offering the functionality the specification describes. Because the relation between specification and implementation is slack, we often want to keep track of the difference between what is actually in the specification itself, and what belongs to one implementation or another. What is written in the specification will of course hold true of all (correct) implementations; it is a requirement. But one implementation may be very different from another. This fact makes it useful to reserve the word "operation" to mean the procedures making up particular implementations, and to use the word "transform" to mean a part of the specification. A transform is the part of a specification describing an operation. Each implementation must have an operation implementing each one of the transforms in the specification. The transform is a piece of text written in Ina Jo describing what an operation must do.

An Ina Jo transform contains two main parts, called a "referential condition" and an "effects section". The referential condition, or "refcond", as the jocular Ina Jo slang calls it, describes an assumption which must hold true in order for the operation to be applied. Thus, it is similar to the "entry condition" of Gypsy and other systems. If the refcond is false, then the effect of the operation is undefined. If the refcond is true, then the effect of the operation is as described in the effects section of the transform. The refcond may be omitted, and often is; then it is counted as being trivially true.

One uses a refcond in order to ensure that an operation will not be applied in circumstances in which it would be illegitimate. For instance, going back to our desk calculator example, it makes no sense to apply the "add" operation if there fewer than two numbers on the stack. After all, "add" pops the top two numbers off the stack, adds them, and places the result on the stack. We might want to make the result undefined iff "add" is invoked when there are fewer than two numbers on the stack. Ina Jo would express this as

```
Refcond (Stack:2) ~= Nil,
```

namely when the part of the stack beginning with the second element is null, the result of the operation is undefined.

The refcond is rather peculiar, and should be used with care. In particular, it is important to realize that it is in no way an exception handling facility. Although one can think of a false refcond as raising an exception, FDM gives us no way of handling the exception. The result of invoking a transform in a state which makes its refcond false is simply to cause the abstract machine to terminate abnormally. We shall return to this point when we are discussing the FDM design verification paradigm. Suffice it to say, for the moment, that if the abstract machine enters any state after applying a transform with a false refcond has been applied, the FDM design verification process will no longer ensure any real correctness. A refcond is most appropriate where there can be some assurance that the transform will never be applied when its refcond is false. Otherwise, it should be used only if the specifier decides that all is lost if the transform is invoked when its refcond is false. The refcond should be thought of as summarizing the conditions which must hold for the transform to make any sense at all.

The examples in this section show slightly bad style, from this point of view. I have used refconds several times, to illustrate the refcond construct, even though I think it would have been better to use a conditional formula in the effect section. I include some examples of this alternate approach in the next section.

The effects section of a transform contains the predicate which describes the state of the variables after the operation. Let us describe the effects section by glossing a few examples. To begin with, the full transform specifying the "add" operation of our desk calculator looks like this.

```
Transform Add        External

     Refcond      (Stack:2) ~= Nil

     Effect
         (  N"Stack = (Stack:3);(Stack.1 + Stack.2)
        &   N"Display =  (N"Stack).1      )
```

The expression External marks a distinction which will not concern us until we turn to multi-level specifications. In a multi-level specification, there is a distinction between External transforms -- the normal kind which describe the operations of abstract machines -- and internal transforms. The latter do not describe full operations; rather they amount to macro definitions. They can be used to simplify the expressions mapping transforms from

an upper level of a specification to a lower level. Internal transforms are conceptually confusing, as one expects a transform to describe an actual operation. It seems likely that the designers of Ina Jo had conflicting goals in mind when they invented the internal transform.

As we noted, the effects section of "Add" is atypical, in that it is deterministic. The new values of the variables are completely determined by it. In this respect, AddTimed is more typical:

```
Transform AddTimed        External

     Refcond      Time >= 0 & (Stack:2) ~= Nil

     Effect
         (  N"Time > Time
        &   N"Stack = (Stack:3);(Stack.1 + Stack.2)
        &   N"Display =  (N"Stack).1      )
```

Our examples so far are still atypical in that they do not have parameters. A transform with a parameter is the specification for our calculator's "push" operation.

```
Transform Push (arg:integer)   External
     Effect
            N"Stack = Stack;arg
        &   N"Display = arg
```

NO CHANGE

Consider next the following context.

```
Type User,
     File,
     Date

Constant
     AccessPermitted(User, File) : Boolean,
     NoOne : User

Variable
     Accessed(User, File, Date) : Boolean,
     CurrentUser(File) : User,
     Today : Date
```

We would like to specify an operation by which a user can open a file, assuming that he is permitted access to it, and no one is

currently using it. The operation should update the Accessed variable, to record that the user has accessed it today, and it should make the user the current user of the file. On the other hand, if the user is not permitted access to the file, or there is already someone using it, then there should be no change in the values of the variables.

Ina Jo contains the convenient "no-change" notation NC"; NC"(var_1, ..., var_n) is shorthand for:

```
  N"var_1 = var_1
&    ...
& N"var_n = var_n,
```

expressing the fact that the new value of the variables are the same as their old value. We will want to use that notation to express the effects in the case where the user is denied access.

Ina Jo also contains an "if then else" notation, which is written

$$P \Rightarrow Q \Leftrightarrow R.$$

This is a logical truth-functional notation, not a procedural notation. It means precisely the same as

$$(P \Rightarrow Q) \ \& \ (\neg P \Rightarrow R),$$

equivalently,

$$(P \ \& \ Q) \text{ or } (\neg P \ \& \ R).$$

It will be important to us to keep in mind that neither of these is the same as the programming language "if then else", which is not a truth functional operator at all. The programming language construct

```
if <expr>
   then <statement1>
   else <statement2>
```

means something entirely different. It instructs us to **do** something, either <statement1> or <statement2>, depending on whether <expr> is true or false. P => Q <> R does not instruct us to do anything, it just expresses a truth value. If these caveats are kept in mind, the notation is valuable. We should, however, stress how important it is to keep these sorts of things in mind; a considerable portion of the specifications for the SCOMP processor

had to be re-done because they treated the truth-functional conditional in HDM as if it were a programming language if-then-else [Platek and Sutherland, 1984].

The apparently simplest way of writing the transform turns out not to be appropriate. One would be tempted to write the effects section in this form:

```
( AccessPermitted (u, f)
 & CurrentUser(f) = NoOne

=>
    N"CurrentUser(f) = u
  & N"Accessed(u, f, Today) = True

<>
    NC"(CurrentUser(f),
        Accessed(u, f, Today) ) )
```

Yet, there is something odd about this predicate. Whether values of CurrentUser and Accessed satisfy the predicate depend only on their results for the arguments f and (u, f, Today) respectively. Thus, it can not constrain the values of CurrentUser(f2) for f2 ~= f, nor those of Accessed(u2, f2, d) where (u2 ~= u or f2 ~= f or d ~= Today). Nor do we know whether the value of Today has changed. This is an unacceptable situation, as we do nct want to end up with an open-file operation which changes the users of other files in an unpredictable way.

On the one hand, we need to ensure that variables do not change value wantonly, while on the other hand it would be too tedious and repetitive to require that every unchanged variable appear with the no-change operator. Clearly, we need some convention.

There are two conventions which one might make on this point. The one that comes to mind first, and which was adopted in SPECIAL, is that a variable like CurrentUser is assumed to be unchanged at an argument f unless the effects section says something about its new value. Thus, in the example above, this convention would tacitly add the information

```
NC"(Today)  &
A" u2 : User, f2 : File, d : Date
   (    ( (u2 ~= u | f2 ~= f | d ~= Today)
          =>
            NC"(Accessed(u2, f2, d))  )
      & ( (f2 ~= f)
          =>
            NC"(CurrentUser(f2) ) )  ).
```

This probably corresponds to the way the reader understood the example as it was written. This convention is sometimes called the "no primed occurrence" convention, as SPECIAL uses the prime sign ´ in much the way that Ina Jo uses the N" operator. (See [VMAN], [Platek and Sutherland, 1984].) The convention states that:

a variable is unchanged at an argument if N" is not applied to the variable with that argument.

Unfortunately, natural as this convention may seem, it is not a good one, as it does not give us a syntactic test. It turns out to be very difficult to express the convention precisely. It does not mean:

> Var(arg) is to be unchanged if the string of characters "N"Var(arg)" does not appear in the effect section.

After all, we may refer to arg other than by its name. Nor will it even do to say:

> Var(arg) is to be unchanged if no string of characters of the form "N"Var(term)" appears in the effect section, where "term" refers to arg.

The notion of "referring" is hard to explicate. After all, consider the following cases:

```
Type IntArray                /* Array of Integers */

Transform ZeroArray (a : IntArray )  External
  Effect
        A"n:integer
          ( N"a(n) = 0)

Transform ZeroEvenPositions (a : IntArray )  External
  Effect
        A"n:integer
          ( Even(n) => N"a(n) = 0)
```

We would have to say that the variable n "refers to" all integers in ZeroArray, and that it "refers to" all even integers in ZeroEvenPositions. It would be difficult to give a definition of "refers to" which would make these examples work out the way we intend.

For these reasons, Ina Jo adopts a different convention. The Ina Jo convention has it that a variable is assumed unchanged -- at all arguments -- if the variable name does not appear anywhere in the effects section with N" prefixed. This is at least a purely syntactic criterion, making it preferable to the "no primed occurrence" convention even though the latter seems more convenient. The Ina Jo convention requires the user to specify more "no-change" cases explicitly, as it does not try to supply as much additional information as the "no primed occurrence" convention. All that the Ina Jo convention allows us to infer from the predicate given above is that the variable Today is unchanged. It is the only variable not occurring anywhere with the new-value sign attached. Therefore, a correct Ina Jo specification for the operation to open a file has to give more explicit information.

```
Transform Open(u:User, f:File) External
  Effect
    ( (AccessPermitted (u, f)
     & CurrentUser(f) = NoOne )

    =>
     (  N"CurrentUser(f) = u
      & N"Accessed(u, f, Today) = True
      & A" u2 : User, f2 : File, d : Date
         ( (u2 ~= u | f2 ~= f | d ~= Today)
         =>
           NC"(Accessed(u2, f2, d) )  )
       & ( f2 ~= f
         =>
           NC"(CurrentUser(f2) ) )  )

    <>
      A" u2 : User, f2 : File, d : Date
         ( NC"(Accessed(u2, f2, d),
              CurrentUser(f2)  ) ) )
```

AN EXAMPLE

Let us work through another example, to examine a few more of the Ina Jo language constructs. Consider the problem of writing a system to keep a common appointment calendar for a group of people. All of them will write their appointments with each other in the calendar. The system uses three main types: Person, DateAndTime, and Appointment. The first two are unspecified -- we will not decide yet how to represent them. An appointment will consist of the set of Persons attending together with the of the meeting. The main variable in the system, called Calendar, takes sets of Appointments as its values.

Thus, in this example we have occasion to use the Ina Jo set operators, which are rather convenient. The operations of set union, intersection, and difference are symbolized by "||", "&&", and "~~" respectively, exploiting the natural analogy between these set operators and the boolean operators symbolized by "|", "&", and "~". Ina Jo also uses the operator **S"** (read: set of) in two slightly different ways. First, **S"(a, ..., b)** denotes the set whose only members are a, ..., b. For instance, **S"(Reagan)** denotes the set whose only member is the President. Second, **S"x:Typename(Pred(x))** denotes the set of all objects x belonging to the type named by Typename such that Pred is true of x. Thus, **S"i:integer(Divisible(i, 2))** denotes the set of even numbers. The fact that an object is a member of a set is expressed with the sign "<:"; e. g., 8 <: S"i:integer(Divisible(i, 2)).

The Ina Jo existential and universal quantifiers **E"** and **A"** are also used; **E"x:Typename (Pred(x))** means that Pred(x) holds for at least one value of x belonging to type Typename. **E"x:Person(Scoundrel(x))** reminds us of the familiar fact that there is at least one person who is a scoundrel. **A"x:Typename (Pred(x))** means that Pred(x) holds for every value of x belonging to type Typename. **A"x:Person(Scoundrel(x))** is a very cynical statement.

The reader should note that this specification uses the Definition facility of Ina Jo. In practice, this is an extremely valuable facility, even though in theory it does not add any expressive power to the language. We want to be able to give a simple name to a complex expression if it will be used at all often.

This sketchy specification contains only three transforms. These have the effect of throwing away all appointments which have already taken place, adding an appointment, and deleting an appointment. Each of the transforms has a refcond. One does not delete an appointment unless it was previously scheduled. One does not schedule an appointment for a time at which one of the participants has another meeting. The refcond of UpdatePresent is a way of noting the direction of time.

I have included the keywords which the Ina Jo language processor expects to surround the specification.

```
$TITLE Cal
SPECIFICATION CalendarKeeper
LEVEL TopLevel
```

```
Type    Person,
        DateAndTime,
        Appointment = Structure of
                        (Who = Set of Person,
                         When =  DateAndTime),
        A_Calendar = Set of Appointment

Constant
        Le (DateAndTime, DateAndTime): Boolean


Axiom  A"d1,d2,d3:DateAndTime (Le (d1, d1)
        & ( ( Le (d1, d2) & Le (d2, d1) ) => d1 = d2)
        & ( ( Le (d1, d2) & Le (d2, d3) ) => Le(d1, d3))
        & ( Le (d1, d2) | Le (d2, d1) ) )
/* This says that Le is really a "less-than-or-equals"
      relation, or a weak linear ordering.  */

Variable Calendar : A_Calendar,
         Present : DateAndTime

Initial
        A"p:Person, a1,a2:Appointment (
                ( a1 <: Calendar & a2 <: Calendar
                & a1 ~= a2 & p <: a1.Who & p <: a2.Who)
                => a1.When ~= a2.When )

        & A"a:Appointment ( a <: Calendar => Le(Present, a.When) )

/* This says that initially, no individual has to be at two
        meetings at the same time, and that no meeting is
        scheduled to take place in the past  */

Define
        Busy(p:Person, t:DateAndTime) : Boolean
                == E"a:Appointment
        (a<: Calendar & p<: a. Who & a. When=t)

        Lt (t1:DateAndTime, t2:DateAndTime) : Boolean
                 == ( Le (t1, t2) & t1 ~= t2 )
                /* strictly earlier than */

        /*************************************/

Transform UpdatePresent (Now : DateAndTime)   External

   Refcond Lt(Present, Now)
```

```
   Effect N"Present = Now
        & N"Calendar = S"a:Appointment
                (a <: Calendar & Le(Now, a.When) )

/* Update the present by assigning a later time to the
        variable Present and dropping all appointments
        which have already taken place   */

        /*************************************/

Transform Adjoin (a : Appointment)              External

   Refcond A"p:Person (p <: a.Who => ~Busy(p, a.When) )
        &  Le(Present, a.When)

   Effect N"Calendar = Calendar || S"(a)

/* adjoin the appointment to the calendar, but only if
        it doesn't create a conflict for a participant,
        and only if it's not scheduled for the past  */

        /*************************************/

Transform Remove (a : Appointment)              External

   Refcond a <: Calendar

   Effect N"Calendar = Calendar ~~ S"(a)

/* remove the appointment from the calendar */

END TopLevel
END CalendarKeeper
```

The axiom on Le is a way of making sure that it will be implemented as a real "less than or equals" relation, i.e. as a weak linear ordering. The initial condition has two parts. The first conjunct of Calendar ensures that no one has conflicting obligations in the initial state of the Calendar. The second conjunct ensures that the Calendar starts out "up to date" in the sense that it does not contain any appointments scheduled to take place in the past. The refcond of Adjoin ensures that no one will be assigned conflicting obligations as the state of the Calendar evolves. The effects section of UpdatePresent, together with the refcond of Adjoin, ensures that the Calendar will remain up to date.

DYNAMIC SEMANTICS

What are the semantics of the dynamic part of a specification? Let us suppose that we already have chosen an interpretation of the static part of the specification: how shall we go on to give an interpretation for the remainder of the specification?

The static interpretation associates a particular structure, or mathematical universe, with the specification. It gives particular sets of objects as the meanings of the type expressions, and fixes objects and functions as the meanings of the individual constants and function constants. Let us call the structure given by the static semantics the "background structure".

The dynamic interpretation will make use of the background structure in order to define an abstract machine. We can then understand the dynamic part of the specification to be talking about this abstract machine. As we shall see, these abstract machines may be non-deterministic.

What does an abstract machine consist in? An abstract machine should contain three parts. First, it must contain a list of variables. Each one of these variables will range over a set included in the background structure. The current values of these variables at any given time (jointly) determine the state of the abstract machine at that time.

Second, the abstract machine contains a particular state, called its initial state. We consider this as the state that it starts off in, every time it is "started up." This initial state is the basis of the definition of the set of "accessible states" of the machine.

Finally, the machine has a set of operations, which cause changes in the current state of the machine. Each operation has a list of parameters, each of which ranges over some sort of object in the underlying structure. When supplied with values for its parameters, and a current state, the operation yields a set of possible resulting states. This set may be empty. If so, the machine cannot pass into a "next-state" after the operation has been invoked with these parameters. This corresponds to a false Refcond. If the set has just one member, then the result of the operation -- when applied to these arguments -- is deterministic. If it has more than one member, then the results of invoking the operation are non-deterministic. Which member of the set becomes the actual next state in one run of the machine is "random".

When is a particular abstract machine a possible interpretation of the dynamic part of a specification? Let us suppose that its underlying structure is a legitimate interpretation of the static part of the spec, according to the approach we worked out in Chapter II, Section 3. Then we want each state variable of the machine to correspond to one of the variables, with a compatible type, in the specification. The initial state of the machine must satisfy the initial condition in the specification.

Moreover, we should be able to correlate the operations of the machine and the transforms in the specification one-to-one. The parameters of a transform should be type-compatible with the parameters of the corresponding operation. Most important, the operation should fit the specifications given in the transform. The set of possible next-states for the operation should be non-empty -- given a current state and a set of parameter values -- just in case the current state and parameter values satisfy the Refcond of the transform. And, when the set is non-empty, then for any one of the possible next-states in the set, the effects section of the transform should be a correct description of the relation between the current state, the parameter values, and the next state.

Let us sketch this out in the case of our calendar example, but without going into complete detail.

The static semantics for our Calendar Keeper can be simple enough. We can interpret the type Person by the set of people belonging to some particular organization, say one department of a company. We will interpret the type DateAndTime by a linearly ordered set. Let us choose the integers. The reader may imagine dates and times correlated with the integers to make the interpretation more useful. The Appointments will correspond to pairs, where the first element -- the "Who" element -- is a set of people, and the second element -- the "When" element -- is an integer. Let us call these pairs "meetings", in order to have a convenient way of talking about them without confusing items in the structure with notations belonging to the specification. The function constant Le will, of course, denote the ordinary less-than-or-equals relation on the integers, so the axiom will turn out true.

The abstract machine we will use to interpret the specification is a deterministic machine, in the sense that for each operation, current state, and list of parameter values, there is never more than one possible next state. Because of this, I will use a possibly dangerous convention in describing it. I will say that an operation is undefined to mean that its result is the null set, and I will say that its result is x when I really mean that its result is the set containing x as its only member.

The machine has just two state variables, though it would be permissible to use more. The first, the Cal variable, will always have a set of meetings as its value. The second, Pres, will have integers as its values.

In the initial state of our machine, Cal will have the null set as its value, and Pres will have the value zero. So our machine will be starting off with no meetings scheduled. This will safely satisfy the initial condition in the specification.

We interpret the transform Update Present by the function UP. UP takes an integer as a parameter. It is defined for an integer n in a state S if the value of Pres in S is less than n. If defined, it yields a state S′ such that:

the value of Pres in S′ is n, and

the value of Cal in S′ is the set of all meetings m such that m belongs to the old value of Cal, and the When component of m is at least n.

UP will be defined for n in state S just if Lt(Present, Now) is true of the value of Pres in S, and n. And the "next state" that the function returns will satisfy the relation described in the effects section of the UpdatePresent transform.

We will interpret Adjoin by a function AM (for "Add Meeting"). AM will take a meeting as a parameter. It should be defined for a meeting m and a state S only if m is not scheduled for some previous time, and adding m to the value of Cal in S would not mean that anyone would have to be in two places at the same time. When it is defined, its result should have the same value for Pres as S had. The new value of Cal should contain m as well as all meetings previously in Cal.

Remove will denote a function Rm taking a meeting as a parameter. Rm should be defined for a meeting m in a state S just if m belongs to the value of Cal in S. The result of Rm should be a new state with the same value of Pres, but with m omitted from Cal.

Interpreting the dynamic part of a specification simply means describing an abstract machine in this way. Of course, we can also use a non-deterministic machine, unlike this one. In that case, we must keep track of the fact that each operation really produces a set of possible next states, rather than a single next state.

SECTION 4

EXPRESSING REQUIREMENTS IN INA JO

CORRECTNESS CONDITIONS

Let us start out from our calendar example in examining the issue of correctness. What does the correctness of the specification involve? It seems to come down to two points. The calendar must not send the same person to two places at the same time, and it should always be up to date, containing only information about the present and the future. Perhaps we should include one other requirement, namely the demand that the direction of time should always flow forward: every time we update the variable Present, the old value should be less than the new value.

As we mentioned earlier, the FDM verification paradigm centers on the two *ass*ertions which FDM terminology calls the criterion and the constraint.

The criterion is a predicate which must hold true in every state that the abstract machine ever enters. In our calendar example, we would want the criterion to assert that no one has conflicting obligations and that the calendar is up to date. It might look like this:

```
Criterion
  A"p:Person, a1,a2:Appointment
       ( ( a1 <: Calendar & a2 <: Calendar
         & a1 ~= a2 & p <: a1.Who & p <: a2.Who)
        => a1.When ~= a2.When )
& A"a:Appointment ( a <: Calendar => Le(Present, a.When) )
```

The constraint describes the changes that any transform can cause: it is a predicate involving both the pre-operation and post-operation values of the variables. For instance, we could express the requirement that time should always flow forward by the constraint

```
Constraint Le(Present, N"Present).
```

If a transform changes the value of the variable Present, then the old value should be less than the new value. (Note that two of the operations do not change its value.)

A specification is correct if, for every machine satisfying the specification, the criterion is true i. every state the machine can ever reach, and the constraint is true for each of the operations of the machine, starting from any state the machine can reach.

Thus to verify a design in FDM, we must formulate a criterion and a constraint, and prove from the specification that they will always be true in any state that the machine will ever enter.

According to our approach to the semantics of an Ina Jo specification, a machine satisfies a specification if its underlying structure satisfies the static part of the specification, and its initial state and set of operations fit the description given in the dynamic part of the specification. Let us suppose given, then, an Ina Jo specification A and a machine M which satisfies it.

We call a state s accessible relative to M if s is the initial state of M, or if there is an operation which may produce s when applied to some list of parameters and a state already known to be accessible.

Or, to restate the idea in an equivalent way, a state s -- that is, an assignment of a value to each of the state variables in M -- is accessible at stage n if:

i) n = 0, and s is the initial state of M,

ii) n = m+1, and there exists a state s1 accessible at stage m, an operation F, and values of the parameters of F, such that s is a member of the set of possible next states F yields when applied to s1 and these values of the parameters.

A state s is accessible if, for some natural number n, it is accessible at stage n.

A specification satisfies its requirements if, for every machine M which satisfies the requirements, any state s which is accessible to M, and any operation F of M, the criterion and the constraint are true for s and F.

How do we go about proving this? What do we know about all the machines satisfying the specification?

Obviously, what we know about all the machines satisfying a specification A is precisely what A says. Thus, we want to use A itself to build a proof about all states accessible to any machine satisfying A.

In fact, this is not hard. We know that the initial state of a machine M satisfying A will verify the initial condition of A. So we must show that the initial condition entails the criterion. This tells us that the criterion will be true in the state accessible to M at stage 0.

And we know that each operation of the machine must satisfy one of the transforms in A. So we must use the transforms to prove that the criterion is satisfied in all states accessible at stage n+1, assuming it was satisfied in all states accessible at stage n. We also use the transforms to prove that the constraint is satisfied when passing from any state accessible at stage n to a state accessible at stage n+1.

Mathematical induction will allow us to conclude that the correctness conditions hold for all accessible states.

How would this work in our calendar-keeper? The argument has a somewhat simpler form in the case of the calendar-keeper than it sometimes does. We will first argue that the criterion is satisfied by all the accessible states; after that, we will argue that each transform satisfies the constraint starting from any state whatever (thus certainly any accessible state).

To begin with, if a state is accessible at stage 0, then it must satisfy the initial conditions on the variables. The initial condition was:

```
Initial
        A"p:Person, a1,a2:Appointment (
                ( a1 <: Calendar & a2 <: Calendar
                & a1 ~= a2 & p <: a1.Who & p <: a2.Who)
                => a1.When ~= a2.When )

        & A"a:Appointment ( a <: Calendar => (Le(Present, a.When) )
```

Now this is -- no coincidence -- identical with the criterion. So all states accessible at stage 0 satisfy the criterion. Moreover, suppose that all states accessible at stage m are already known to satisfy the criterion; we want to prove that the states accessible at the next stage will still satisfy the criterion.

Suppose that s is accessible at stage m, so we know that s satisfies the criterion. And suppose that a transform T could transform s into a new state s´. We want to show that s´ still satisfies the criterion. We argue by cases, depending on which transform T is.

1. T is UpdatePresent:

```
Transform UpdatePresent (Now : DateAndTime)   External

   Refcond Lt(Present, Now)

   Effect N"Present = Now
          N"Calendar = S"a:Appointment
             (a <: Calendar & Le(Now, a.When) )
```

We must show that in the new state s', both halves of the criterion are satisfied. That is, no one should have two meetings scheduled for the same time, and no meeting should be scheduled for a time already past. Now the second conjunct of the effects section ensures that every appointment in the new calendar was already in the old version of the calendar. So if no one had a conflict before the transform was applied, no one can have a conflict after the transform was applied. Moreover, the effects section also ensures that no appointment will remain in the calendar unless it is scheduled for Now or a time after Now. Therefore both parts of the criterion will be satisfied after the transform if they were satisfied before.

2. T is Adjoin:

```
Transform Adjoin (a : Appointment)            External

   Refcond A"p:Person (p <: a.Who => ~Busy(p, a.When) )
        &  Le(Present, a.When)

   Effect N"Calendar = Calendar || S"(a)
```

Again we must show that both halves of the criterion will be satisfied. In this case, the Effect section just tells us that the appointment a is the only one we really have to worry about. The Refcond contains the crucial pieces of information. We know that the first half of the criterion will not be ruined, because the Refcond tells us that no one in meeting a had any other obligation for that time. And the second half of the Refcond tells us that the new meeting has not been scheduled for the past.

3. T is Remove:

```
Transform Remove (a : Appointment)            External

   Refcond a <: Calendar

   Effect N"Calendar = Calendar ~~ S"(a)
```

In this case, there is nothing to show, as the effects section ensures that the new calendar is included in the old calendar. Thus if the criterion held in the old state, it must hold in the new state.

We can also easily check that the constraint

Constraint Le(Present, N"Present)

will always be satisfied. Since the only transform which changes the value of the variable Present is UpdatePresent, and since the Refcond of UpdatePresent is Lt(Present, Now), the constraint is surely satisfied.

So we have given an intuitive argument for the correctness of our calendar-keeper. The argument is simpler than it might have been, in two ways. First, we could prove that the constraint would be satisfied without worrying about what sort of state we were starting out from, whether an accessible state (which would satisfy the criterion) or not. In general, the argument for the constraint may make use of the assumption that we are working with an accessible state.

Actually FDM gives us slightly more flexibility than this. It is a peculiarity of proof by induction that it is often easier to prove a stronger statement instead of a weaker statement. The reason is that in proof by induction, we argue that our conclusion is true for m+1 on the basis of the assumption that it is true for m. Sometimes we need a strong assumption. For instance, suppose we define a function f on non-negative integers by saying:

$$f(0) = 10;$$

$$f(n+1) = \begin{cases} f(n) + 1 & \text{if } f(n) > 8 \\ f(n) - 1 & \text{otherwise.} \end{cases}$$

Now if we want to prove that f(n) is always greater than 0, we will actually have to prove something stronger. Namely, we will have to show that f(n) is always greater than 8. This is true because

i) $f(0) > 8$; and

ii) if $f(n) > 8$, then $f(n+1) = f(n) + 1 > f(n) > 8$.

Since f(n) is greater than 8, it is also greater than 0. The reason we need to prove the stronger assertion to get the weaker one is that we need to assume that $f(n) > 8$ to infer that $f(n+1) > f(n)$.

Thus step ii) collapses if we are trying to prove directly that f(n) is always greater than 0.

This situation arises often when one is arguing by induction. Thus FDM allows us to prove something stronger than we really need. We may add an "invariant condition" to our criterion. The invariant condition, like f(n) > 8 in the example above, is an additional claim which may not interest us in itself at all. Its only purpose is to help us complete the inductive argument by summarizing information which must remain true for the argument to go through.

We can now express the theorems which FDM requires us to prove for correctness. We must first prove that the criterion and the additional invariant condition will start out true -- that they are true in all states accessible at stage 0. Since all we know about the states accessible at stage 0 is that the initial conditions on the variables are true, we must prove:

```
Initial-condition => Invariant-condition & Criterion
```

To complete the proof of correctness, we have to show that the correctness conditions will hold true of any state accessible at stage m+1, assuming that they held true for all states accessible at stage m. To show this, we must show, for each transform T in the specification, that if we apply T to a state in which the correctness conditions are satisfied, then they must remain satisfied. For if no transform in the specification allows correctness to be destroyed, all states accessible at the next stage must be correct. Thus for each transform T, we want to be proving:

```
Invariant-condition & Criterion & T
                => N"Invariant-condition  & N"Criterion
and
Invariant-condition & Criterion & T
                              => Constraint
```

By N"Expr I mean the expression one gets by replacing each state variable Var by N"Var throughout Expr. For instance, if Expr is

```
A"p:Person, a1,a2:Appointment
     ( ( a1 <: Calendar & a2 <: Calendar
       & a1 ~= a2 & p <: a1.Who & p <: a2.Who)
      => a1.When ~= a2.When ),
```

then N"Expr is

```
A"p:Person, a1,a2:Appointment
```

```
( ( a1 <: N"Calendar & a2 <: N"Calendar
  & a1 ~= a2 & p <: a1.Who & p <: a2.Who)
 => a1.When ~= a2.When )
```

Calendar, the only state variable occurring in the expression, has been replaced by N"Calendar. Thus, N"Invariant-Condition says that the invariant condition holds of the new values of the variables, and N"Criterion says that the Criterion holds of the new values. The Constraint, of course, describes the relation between the new and old values of the variables, so we do not have to apply the N" operator to it.

But what do we mean by inserting the transform T as an assumption? The transform is represented by the conjunction of its refcond and effects section. Thus for instance, the first theorem to be proved for the transform Adjoin would have the form:

```
  Invariant-condition & Criterion

& A"p:Person (p <: a.Who => ~Busy(p, a.When) )
        &  Le(Present, a.When)          /* (Refcond)*/

& N"Calendar = Calendar || S"(a)
                        /*(Effect)*/

=>

  N"Invariant-condition  & N"Criterion,
```

where Invariant-condition and N"Invariant-condition happen to be vacuous. And the argument, showing that the criterion is still true after Adjoin has been applied, relied on both the Refcond and the Effect section.

Because Refconds are used as premises in the proof that a specification is correct, the FDM verification paradigm tells us nothing about correctness in case a transform is ever applied when its Refcond is false. Everything that we prove presupposes that they are always true when the transforms are applied. Therefore, we cannot allow the abstract machine to pass into any state at all if a transform is applied when its Refcond is false. For if the machine passed into any state, we would have to know that the resulting state satisfied the correctness conditions. Since the proofs of correctness assume the truth of the Refcond, we would have no information about the resulting state. This is the reason the Refcond is in no way an exception handling mechanism: if an "exception" of this sort is ever raised, the machine terminates abnormally.

Let us call this the "no next state problem".

The reader might think it easy to get around this problem. One might suggest that if a transform is applied when its refcond is false, then there should be no change in the value of any variable. Since the state will be unchanged, the machine cannot pass from a correct state into an incorrect state.

Strictly speaking, this interpretation is not open to us. It is perfectly legitimate to have a constraint requiring that every transform should change the value of at least one variable. And we may be able to prove the correctness theorems

```
Invariant-condition & Criterion & T
                              => Constraint.
```

Yet if T contains a refcond, this contradicts the proposed interpretation. We will have proved that there is always some change. So the interpretation is not acceptable as it stands.

Nevertheless, this contains the germ of the right solution to the no next state problem, namely that the Refcond is no longer needed. Instead of

```
Transform {Name and parameters}                External

    Refcond {Formula1}

    Effect  {Formula2},
```

we can always write:

```
Transform  {Name and parameters}               External

    Effect   {Formula1}
             =>
              {Formula2}
             <>
              NC"({variables}).
```

As the latter is what we mean, it is much to be preferred. Thus we have a simple translation strategy to eliminate the Refconds from Ina Jo specifications. The resulting specification will ensure the same behaviour so long as no transform is invoked with a false Refcond. When a transform is invoked when its Refcond is false, there will be no state change. Specifications in this form do not present us with the "no next state problem".

CORRECTNESS AND SECURITY

Let us now look more closely at the way the conception of correctness embodied in FDM influences the designer of a secure computing system. We will begin by working through a specification which embodies a simplified Bell-LaPadula [Bell and LaPadula, 1975; Millen and Cerniglia, 1984] approach to computer security.

The Bell-LaPadula model divides the entities that it deals with into subjects and objects. A subject is an "active" entity, in general either a user or a process, although in our simplified example we can think of each subject as a user. An object is a "passive" entity, a repository of data, like a file. Each object has a security level, which one generally takes to be a classification (unclassified, confidential, secret, top secret) together with a set of need-to-know categories, e.g. nuclear or crypto. A user has a right to read a file only if he is cleared to its level and fits all its need-to-know compartments. Although this is the usual interpretation of security levels in practice, for our purposes, all we need to know about the security levels is that they are partially ordered.

Each user has a maximum security level, the level he is cleared to, and also a current level. Since the basic maxim of the Bell-LaPadula model is "no read up, no write down", a user's current level must dominate the levels of all the files he is currently reading, and must in turn be dominated by the levels of any files he is currently writing to. This ensures that no information can pass from a file into another, less highly classified file. The requirement that no user read a file above his current level is called the "simple security condition". The requirement that he not write into a file below his current level is called the "*-property". Bell and LaPadula also require that the level of a file not change during normal operation.

Besides subjects, levels, and files, our specification has a type for file identifiers, one for the items to be stored in files, and one to represent the kind of access that a user has to a file (read or write).

```
$TITLE B_L_1
SPECIFICATION B_L_version1
LEVEL TopLevel

Type    Subject,
        Level,
        FileId, /* items suitable for naming
             files, perhaps character strings */
```

```
        Content,  /* whatever items we
               want to store in files */
        Cont_List = List of Content,

        Accesskind = (read, write),

        File = Structure of (flevel = Level,
                             fowner = Subject,
                             fcontents = Cont_List)

Constant
        SysLo : Level,
          /* Lowest level on system */

        lteq(Level,Level) : Boolean,
          /* The partial ordering of security levels */

        Clearance(Subject) : Level,
          /* User's maximum level */

        NoOne : Subject
          /* Owner of files not currently in use */

Axiom
        /* lteq is a partial ordering with least element SysLo */

        A"l1,l2,l3:Level
         (
            lteq(l1, l1)
        & ( lteq(l1, l2) & lteq(l2,l1) => l1 = l2)
        & ( lteq(l1, l2) & lteq(l2,l3) => lteq(l1, l3))
        &   lteq(SysLo, l1)
          )
```

The state of the system is determined by six variables. Although each one of them is specified as if it were a large array, some of them would certainly have to be implemented in other ways. The role of the first five variables is to record the current state of the file system, indicating who has the right to access files, and who is currently accessing them. The last variable, Buffer, contains the information that users are currently reading out of the file system or writing into it.

```
Variable
        InUse(FileId) : Boolean,
          /* True if  Id is currently associated
                    with a real file*/
```

```
        FileSystem(FileId) : File,
          /* "Retrieves" the file given its id */

        Discretion(Subject, FileId, Accesskind) : Boolean,
                /* Indicates owner intends to allow subject
                        accesses of this type */

        AccessMatrix(Subject, FileId, Accesskind) : Boolean,
                /* Indicates subject has file open for accesses
                        of that type */

        CurrentLevel(Subject) : Level,
          /* Must be higher than the files currently
                being read, lower than the files
                currently being written */

        Buffer(Subject, Accesskind) : Cont_List
                /* Subject's read buffer is like his monitor;
                        write buffer is like his keyboard */

Define
          /*  A few convenient abbreviations  */

   ReadOk(s:Subject, f_id:FileId) : Boolean

                == lteq(FileSystem(f_id).flevel,
                                        CurrentLevel(s)),
             /* SimpleSecurity */

   WriteOk(s:Subject, f_id:FileId) : Boolean

                 == lteq(CurrentLevel(s),
                                FileSystem(f_id).flevel),
               /* StarProperty */

   AccessOk(s:Subject, f_id:FileId, a:Accesskind) : Boolean

         ==    InUse(f_id)
            & (a = read => ReadOk(s,f_id) )
            & (a = write => WriteOk(s,f_id) )
            & ( Discretion (s, f_id, a) )
```

The correctness conditions for our specification are the requirements in the Bell-LaPadula model. They lay down the simple security property and the *-property, in addition to the discretionary policy that individuals choose for the use of their files. Moreover, the current level of a user cannot exceed his clearance. One additional requirement concerns files that are

discarded: their contents must be destroyed. These conditions make up the criterion for our specification. The constraint is that the level of a file should not change while it is in use.

```
Initial  /* CurrentLevel le clearance & AccessOk
            & Non-Existent Files Are Null */

        A" s : Subject, f_id : FileId, a : Accesskind
         ( lteq(CurrentLevel(s), Clearance(s))

        &  (AccessMatrix(s, f_id, a)
                => AccessOk(s, f_id, a) )

        &  (~InUse(f_id) => FileSystem(f_id).fcontents = Nil
                        & FileSystem(f_id).fowner = NoOne
                        & FileSystem(f_id).flevel = SysLo)
                /* Clear Out Unused Objects */
          )

Criterion   /* CurrentLevel le clearance & AccessOk
            & Non-Existent Files Are Null */

        A" s : Subject, f_id : FileId, a : Accesskind
         ( lteq(CurrentLevel(s), Clearance(s))

        &  (AccessMatrix(s, f_id, a)
                => AccessOk(s, f_id, a) )

        &  (~InUse(f_id) => FileSystem(f_id).fcontents = Nil
                        & FileSystem(f_id).fowner = NoOne
                        & FileSystem(f_id).flevel = SysLo)
                /* Clear Out Unused Objects */
          )

Constraint /* files in use retain their levels: tranquillity */


        A"f_id : FileId (
        ( InUse (f_id) & N"InUse(f_id) )
                => FileSystem(f_id).flevel = N"FileSystem(f_id).flevel
           )
```

Next we must specify the operations we want our machine to perform. The specification will contain seven transforms. A subject can get access to a file, or close it; he can read it or write it if he has the right kind of access; he can create a file or, if he has write access, destroy it. A subject can also change his current level.

The GetAccess transform causes no change if the subject would not be permitted to have the requested kind of access to the file in question. If the access is permissible, it updates the AccessMatrix. DiscardAccess(s, f_id, a) causes AccessMatrix(s, f_id, a) to be false.

```
Transform GetAccess(s : Subject,
                    f_id : FileId,
                    a : Accesskind)            External

   Effect
   (
     AccessOk(s, f_id, a)
     =>
           N"AccessMatrix(s,f_id,a) = True

         & /* No other entry is changed */

           A" s2:Subject, f2:FileId, a2:Accesskind(
               ( s2 ~= s | f2 ~= f_id | a2 ~= a )
                 =>
                 NC"(AccessMatrix(s2,f2,a2) )
                 )

      <>   /* Else no change */

         A" s2:Subject, f2:FileId, a2:Accesskind
            (NC"(AccessMatrix(s2,f2,a2)  ) )
      )

Transform DiscardAccess(s : Subject,
                        f_id : FileId,
                        a : Accesskind)        External

   Effect
        (
              N"AccessMatrix(s,f_id,a) = False

          &  /* No other entry is changed */

              A" s2:Subject, f2:FileId, a2:Accesskind
               (
                ( s2 ~= s | f2 ~= f_id | a2 ~= a )
                  =>
                  NC"( AccessMatrix(s2,f2,a2) )
               )
        )
```

What can a subject do with a file he has access to? The specification will contain an operation to read it into his read buffer, and an operation to overwrite the file with the contents of his write buffer.

```
Transform ReadOp(s : Subject,
                 f_id : FileId)                External

    Effect
        (
        AccessMatrix(s, f_id, read)
        =>
        /* Concatenate the contents of the file with the current
                       contents of the buffer */

             N"Buffer(s, read) =
                  Buffer(s, read);;FileSystem(f_id).fcontents

           & /* No other buffer is changed */

             A" s2:Subject, a:Accesskind
               (
                (s2 ~= s | a ~= read)
                => NC"(Buffer(s2, a) )
                )
        <> /* Else no change */

           NC"(Buffer)
         )

Transform WriteOp(s : Subject,
                  f_id : FileId)               External

    Effect
        (
         AccessMatrix(s, f_id, write)
        =>
             N"FileSystem(f_id)
                = (FileSystem(f_id).flevel,
                   FileSystem(f_id).fowner,
                   Buffer(s, write) )

           & /* No other entry is changed */

             A" f2:FileId, a:Accesskind
                (f2 ~= f_id
                 => NC"(FileSystem(f2) ) )
        <> /* Else no change */

           NC"(FileSystem)

        )
```

This specification will not go into any more detail about the read and write buffers and what we can do with them. Nevertheless, the reader can imagine operations that any implementation of interest would provide. For instance, we should be able to manipulate the contents of the read buffer, and also pass selected contents from the read buffer into the write buffer.

One can create a file with a given FileId only if the identifier is not already in use. If the identifier is not in use, the transform causes it to come into use, and to be associated with a file. The file has null contents. Its owner is the subject who created it; its security level is specified by a parameter. On the other side, one can destroy any file to which one has write access. When a file is destroyed, its identifier is taken out of use, and comes to be associated with a null file.

```
Transform Create (s : Subject,
                  f_id : FileId,
                  l : Level)                    External

     Effect
        (
         ~InUse(f_id)
        =>
          /* The changes */

            ( N"InUse(f_id) = True

            & N"FileSystem(f_id).fowner = s
            & N"FileSystem(f_id).flevel = l
            & N"FileSystem(f_id).fcontents = Nil

            & /* Discretionary policy can be anything */

              A"s2:subject, a2: Accesskind
               (N"Discretion(s2, f_id, a2)
                = N"Discretion(s2, f_id, a2) )

            & /* No other entry is changed */

              A"f2:FileId (f2 ~= f_id
                => ( NC"(InUse(f2) )
```

```
                & NC"(FileSystem(f_id))
                & A"s2:subject, a2: Accesskind
                     (NC"(Discretion(s2, f2, a2) ) )
                          )
             ) )

        <>  /* Else no change */
            ( NC"(InUse, FileSystem, Discretion) )
        )

Transform Destroy(s : Subject,
                  f_id : FileId)                External

    Effect
        (
          AccessMatrix(s, f_id, write)
        =>
             N"InUse(f_id) = False

           & N"FileSystem(f_id).fcontents = Nil
           & N"FileSystem(f_id).fowner = NoOne
           & N"FileSystem(f_id).flevel = SysLo

           & /* Delete all accesses to destroyed object */

             A"s2 : Subject, f2 : FileId, a : AccessKind (
                  N"AccessMatrix(s2, f2, a)
                <->   f2 ~= f_id
                    & AccessMatrix(s2, f2, a) )

        <> /* Else no change */
           (NC"(InUse, FileSystem, AccessMatrix) )
        )
```

The final transform allows a user to change his current level. As a side effect, it empties his read and write buffers. This ensures that when a user lowers his current level, no highly sensitive information can tag along. However, the change of level is permitted only if the new level is dominated by the user's clearance, and only if the new level and the files the user has open satisfy the simple security property and the *-property. Otherwise, there is no change.

```
Transform ChangeLevel(s : Subject,
                      l : Level  )              External

   Effect
        (   /* check new level preserves criterion */
```

```
            lteq(l, Clearance(s) )
          & A"f:FileId (
             (AccessMatrix(s, f, read) =>
                      lteq(FileSystem(f).flevel, l))
            &(AccessMatrix(s, f, write) =>
                      lteq(l, FileSystem(f).flevel))
                      )
        =>
            N"CurrentLevel(s) = l
          & N"Buffer(s, read) = Nil
          & N"Buffer(s, write) = Nil  /* flush buffers */

          & /* No other levels or buffers change  */

            A"s2:Subject, a:AccessKind
               ( s2 ~= s
                 => NC"(CurrentLevel(s2),
                        Buffer(s2, a) )  )

        <> /* Else, no change */
            NC"(CurrentLevel) & NC"(Buffer)
        )

END TopLevel
END  B_L_version1
```

There is an important defect to this specification. It is not secure. There is a gaping-wide storage channel. But before discussing the channel, and seeing how to close it, let us check that the specification satisfies its correctness conditions.

The constraint, which says that the level of a file does not change while the file is in use, is clearly satisfied. For, the only transforms which can change the level of a file are Create and Destroy. Moreover, Create produces no change if applied to a FileId already in use, while Destroy causes a FileId no longer to be in use, when it causes any change. Therefore no transform can apply to a FileId which is in use and change the level of the file it is attached to, while leaving the FileId in use.

As the initial condition is the same as the criterion, we are assured that the criterion will be true at all states accessible at stage 0, which is to say, all states the machine can start off from.

Suppose, next, that the machine is in a state satisfying the criterion. We want to show that its next state must also satisfy the criterion. The criterion is a conjunction of three clauses. We will argue that none of them can become false in the transition to the next state.

The first, namely:

```
lteq(CurrentLevel(s), Clearance(s)),
```

can only be affected if the transform applied is ChangeLevel. But the CurrentLevel of s is changed to l only if

```
lteq(l, Clearance(s)).
```

Thus this conjunct cannot become false.

The only transform which could make the second conjunct of the criterion,

```
AccessMatrix(s, f_id, a)
=> AccessOk(s, f_id, a),
```

false would be GetAccess. But GetAccess makes N"AccessOk(s, f_id, a) true only if either AccessOk(s, f_id, a) was already true, or else AccessOk(s, f_id, a), is true. Thus the second conjunct must also be preserved.

Finally, the third conjunct,

```
~ InUse(f_id) => FileSystem(f_id).fcontents = Null
       & FileSystem(f_id).fowner = NoOne
       & FileSystem(f_id).flevel = SysLo
```

is sensitive only to the transform Destroy. And destroy actually resets the contents, owner and level associated with any FileId it takes out of use.

Therefore the criterion must be preserved as we pass to the next stage. By mathematical induction, we can infer that the criterion will be true in every accessible state.

Nevertheless, the specification is not secure. The problem is in the variable InUse. A user currently at a high level can, in effect, write information into InUse by creating files, that is, by causing certain FileIds to come to be in use. These FileIds may be assigned low security levels. Thus, a user operating at a lower level can access the files. His success or failure in accessing one of these files in effect allows him to read a bit of data written by the high level user. Thus, the transform Create allows the high level user to write down, into the variable InUse.

It is also possible to read up using the same channel. To discover whether a file with a given name is in use at a higher level, the lower level user tries to create it with his own level. If it was not in use, he will be able to access the resulting file. If it was already in use, he will not be able to access it.

This problem, once recognized, is easily solved. In essence, the idea is to stratify the type FileId, so that a file identifier will carry its own level with it. We will then be able to use the same name at different levels without any conflict. We will also treat the transform Create as an operation which both reads and writes information. Thus, following the Bell-LaPadula principles, we will allow the user to create a file only at his current level. For, the level of the file must not be below the current level, as that would amount to a write-down; nor can it be above the current level, lest there be a read-up.

We give now those parts of the revised specification which are not completely predictable.

```
TopLevel B_L_version#2

Type    Subject,
        Level,
        Content,
        Cont_List = List of Content
        FileName,  /* Just the Name */
        Accesskind = (read, write),
        Descriptor = Structure of  (flevel : Level,
                                    fname : FileName),
            /* The real structure identifying the file */

        File = Structure of (fowner : Subject,
                             fcontents : Cont_List )

Variable
        InUse(Descriptor) : Boolean,
          /* True if Descriptor is currently associated
                    with a real file*/

        FileSystem(Descriptor) : File,

        Discretion(Subject, Descriptor, Accesskind) : Boolean,
                /* Indicates owner intends to allow subject
                        accesses of this type */

        AccessMatrix(Subject, Descriptor, Accesskind) : Boolean,
                /* Indicates subject has file open for accesses

                        of that type */

        CurrentLevel(Subject) : Level,
```

```
        Buffer(Subject,Accesskind) : Cont_List
                /* Subject's read buffer is like his monitor;
                        write buffer is like his keyboard */

Define

   ReadOk(s:Subject, d:Descriptor)

        == lteq(d.flevel, CurrentLevel(s))
                                        /* SimpleSecurity */

   WriteOk(s:Subject, d:Descriptor)

        == lteq(CurrentLevel(s), d.flevel)
                                          /* StarProperty */

   AccessOk(s:Subject, d:Descriptor, a:AccessKind)

        == InUse(d)           /* Can't access unused d */
        & (a = read => ReadOk(s,d) )
        & (a = write => WriteOk(s,d) )
        & ( Discretion (s, d, a) )
                                /* DiscretionaryAccess */

Transform Create (s : Subject,
                  d : Descriptor)                External

     Effect (
             ~InUse(d)
           & d.flevel = CurrentLevel(s)
        =>
          /* The changes */
             N"InUse(d) = True
           & N"FileSystem(d).fowner = s

           & A"s2:subject, a2: Accesskind
              (N"Discretionary(s2, d, a2)
               = N"Discretionary(s2, d, a2) )
        /* Discretionary policy can be anything */

           & A"f2:Descriptor, s2:subject, a2: Accesskind
             (f2 ~= d
               =>  NC"(InUse(f2),
                       FileSystem(d)
                       Discretionary(s2, f2, a2) )
                              )
          /* No other changes */
```

```
<>      /* Else no change */
   NC"(InUse, FileSystem,
        Discretionary)
)
```

There is, I think, an important lesson to be drawn from this example: one must be careful in interpreting the significance of the theorems one proves about one's specifications.

Since we succeeded in showing that the first version of our specification satisfied the criterion and constraint, there is a sense in which we can say that the first version was proved to satisfy Bell and LaPadula's model. The criterion and constraint transcribe the Bell-LaPadula axioms. Nevertheless, this would be misleading. When we argued that InUse and Create formed a covert channel, we were in effect arguing that we could use them to write information down to a lower level or to read information from a higher level. Thus, there is also a sense in which the first specification violated the Bell-LaPadula model.

Our proof of the criterion and the constraint established that the Bell-LaPadula model is satisfied only under an artificial interpretation. The proof convinces us only if we assume that every action which amounts to "reading" is an instance of ReadOp, and every action which amounts to "writing" is an instance of WriteOp. That is unrealistic. We are writing whenever we store information in the system. We are reading whenever we recover that information.

This cautionary fable suggests that it can be difficult to tell whether an FDM criterion and constraint really express the correctness conditions we want them to express. This, I believe, is an intrinsic problem with the model of correctness embodied in FDM.

LIMITS TO THE FDM MODEL OF CORRECTNESS

Although the FDM correctness paradigm is flexible, there are definite limits to its expressive power. Design verification in FDM means formulating one's requirements as a criterion and a constraint, and then proving from the specification that they will always be true in any accessible state. This is a somewhat specialized notion of correctness.

The logical form of the statement that a specification is correct is:

(C) For every state S the system can ever enter, and every operation T,

A. S satisfies the Criterion, and

B. S, together with the state which results when we apply T to S, satisfies the Constraint.

This means that it makes sense to partition the individual states into two classes, good states and bad states, where we count a state s1 as good if (C) holds for the particular state s1. Correctness means that every state the system can ever enter will be a good state. The FDM correctness theorems say in essence that every accessible state is good. There is no a priori reason to think that this is the only notion of correctness that we might want. Moreover there actually are cases where we want to use a notion of correctness quite different in form.

One rather different conception is the isolation model, which, although it was suggested at SRI, was never integrated into HDM. The SRI idea [Goguen and Meseguer, 1982, 1984], in brief, is that the machine will receive a stream of commands, each command having a security level attached. If s is a stream of commands, and l is a security level, let's say that another sequence of commands t "sanitizes s at level l" if t contains just those commands from s which have levels less than or equal to l. The system will count as secure if, for each stream s and security level l, the output of the commands of level l is the same whether they are submitted in the stream of commands s or in its sanitization t. In other words, what the system does with the commands of level l does not depend on whether any higher level commands have been executed or not.

While this model of security is simple and plausible, it is not expressible in the FDM paradigm. For, correctness in the FDM sense means that every state that the machine can ever enter will satisfy a criterion and a constraint. Correctness in the sense of the SRI model has a different logical form. It asserts something about every stream of commands, and does not discuss the individual states that the machine enters.

Thus FDM has "wired in" a particular interpretation of correctness, which is by no means the only one which might interest us.

Moreover, it is not exactly the notion of correctness we want for applications to computer security. FDM does not give us a way of asserting that a specification allows only secure information flows. Information flow analysis on FDM specifications has not always been done, for this very reason. Only a tentative information flow analysis was done for KVM, and that was not done by SDC at all, but at MITRE, using a MITRE-developed FDM information flow tool [Huff, 1981; Kramer, 1981, 1982]. However, this special-purpose information flow tool would not have been necessary had FDM been able to express the correctness conditions which are our primary concern.

HDM is really in the same situation as FDM, at least from this point of view. The correctness paradigm of HDM is also oriented toward dividing the possible states of an abstract machine into acceptable and unacceptable classes, and proving that the machine will never enter a state in the unacceptable class. The developers of HDM found it necessary to build a special, add-on, information flow tool, to test whether HDM specifications meet their requirements. This flow tool, while quite useful, is nevertheless inflexible. The security policy it enforces is built into the software itself, and is not stated as an explicit part of the specification. This problem, which is apparently intrinsic to flow tools, certainly affects the MITRE Ina Jo flow tool also. For this reason, information-flow tools represent only an interim solution in the search for expressive specification/verification systems.

SECTION 5

MULTI-LEVEL SPECIFICATIONS

INTRODUCTORY

An important element of the FDM paradigm, although it is perhaps over-emphasized by the developers, is the idea of hierarchical levels of description. Because a specification for a complex system does not leap fully formed from the brow of the designer, like Athena from the brow of Zeus, one of the leading ideas in the development of FDM was to allow the designer to begin working with highly partial specifications. An iterable process would lead from a "system sketch" to a more complete overview of the operations available in the final machine, and a more detailed representation of the data structures they will work on.

There are a number of advantages to this incremental development. First comes the fact that the partial sketch can be subjected to discussion and critical evaluation at an early stage in the design process. This allows the designer to build a consensus that his design offers the right functionality, allowing him to canvass all the alternative approaches at an early stage, before the effort he has already expended makes him unsympathetic to them. Second, the sketches remain valuable at all stages of the development process, as they provide a layered picture of the system being designed. They introduce the main outlines before the details, and illuminating details before more trivial ones. The layered development process encourages the designer to build up the system in a rational way, and helps others to understand its structure.

A third type of advantage makes life easier for the theorem prover, as well as for the human beings. The developers of FDM intend us to prove the theorems guaranteeing the correctness of the specifications in stages, corresponding to the layers in the specification. We prove, to begin with, that what appears in the top layer satisfies the system requirements. We must prove, of each successive layer, both that it is faithful to its predecessors, preserving their adherence to the system requirements, and also that any new operations it adds also satisfy the system requirements. Instead of handing the theorem prover one enormous theorem to establish, we can give it a sequence of more moderate sized claims. This is an important consideration in any system. It becomes an overriding issue in FDM, as the theorems generated by the Ina Jo language processor are already too large in many cases, a problem to which we will return.

An Ina Jo specification consists of a sequence of "levels". The top level contains transforms describing only the most important operations of the final machine. These transforms may be highly nondeterministic, giving only an abstract view of the effect of the operations. Moreover, the variables and constants appearing at the top layer will belong to abstract types, often constructed as sets or sequences from unspecified base types. Each level (except the last) is connected to its successor by a "mapping". A mapping is a correlation between expressions of the upper or "parent" level and expressions of the lower, "child level".

In the discussions which follow, I will not generally talk about more than two levels. I will simply speak of an upper level and a lower level. The reader should keep in mind that the upper level need not be the top level of the whole multi-level specification, nor need the lower level be at the bottom. It will be enough to describe what goes on in any link along the chain from top to bottom.

Each name for a type on the parent level must be matched up with a type on the lower level. Each parent level variable must be matched up with an expression on the lower level, which "represents" the variable in terms of one or more lower level variables. Similarly for upper level constants and transforms.

CONTINUING OUR EXAMPLE: MAPPING TYPES, CONSTANTS AND VARIABLES

Let us look at one way of giving a second level for the calendar example of Chapter III. First, we may want to add a little more structure to the data types:

```
$TITLE Cal
SPECIFICATION CalendarKeeper
LEVEL TopLevel

/***  The Example of Section 3 would be placed here  ***/

END TopLevel

LEVEL SecondLevel UNDER TopLevel

Type    Person,
        Group = Set of Person,

        ChString,       /* For displaying info to user */
        Message = List of ChString,
```

```
        Date,
        Time,
        Occasion = Structure of
                        (WhatTime = Time,
                         WhatDay = Date ),

        Appointment = Structure of
                        (Who = Group,
                         When = Occasion),

        CalendarType = Set of Appointment

Map

        Person == Person,
        DateAndTime == Occasion,
        Appointment == Appointment
```

We have expanded DateAndTime into a structure which contains both a date and a time, and we have added the types ChString and Message so that we can specify some replies to the user.

We will also need some function constants and a few character string constants.

```
Constant
          /* weak linear orders */
        Led (Date, Date) : Boolean,
        Let (Time, Time) : Boolean,

          /* functions to  write messages */
        WriteTime(Time) : ChString,
        WriteDate(Date) : ChString,
        WriteName(Person) : ChString,
        WriteAppointment(Appointment) : ChString,

          /* some important expressions */
        disengaged, uninvited,
        cancelled, nonexistent, OK, impossible : ChString

Axiom
          /* Led is a weak linear ordering */
        A"d1,d2,d3:Date (Led (d1, d1)
         & ( ( Led (d1, d2) & Led (d2, d1) ) => d1 = d2)
         & ( ( Led (d1, d2) & Led (d2, d3) ) => Led (d1, d3) )
         & ( Led (d1, d2) | Led (d2, d1) ) )
```

```
        /* Let is a weak linear ordering */
    & A"t1,t2,t3:Time (Let (t1, t1)
       & ( ( Let (t1, t2) & Let (t2, t1) ) => t1 = t2)
       & ( ( Let (t1, t2) & Let (t2, t3) ) => Let (t1, t3) )
       & ( Let (t1, t2) | Let (t2, t1) ) )

       /***** We could also require that the "write" */
       /*   functions are one-to-one.  In practice,  */
       /*   they must be, so that output information */
       /*   is unambiguous.                    ******/

Define
       Busy(p:Person, o:Occasion) : Boolean

        == E"a:Appointment
             (a <: Calendar & p <: a.Who & o = a.When),

         /* weak ordering on occasions */
       Leo (o1:Occasion, o2:Occasion) : Boolean

        ==  (  o1.WhatDay = o2.WhatDay
             & Let(o1.WhatTime , o2.WhatTime) )
               /* Earlier same day */

          | (  o1.WhatDay ~= o2.WhatDay
             & Led(o1.WhatDay,  o2.WhatDay) )
               /* Previous day */
```

Using these constants and definitions, we can give the mapping of the upper level constant.

```
Map
       Le(d1, d2) == Leo (d1, d2)
```

What type does the variable d1 range over? Since Le is declared in the upper level:

```
   Constant
        Le (DateAndTime, DateAndTime): Boolean,
```

the variable d1 must be ranging over the type DateAndTime. But since Leo is defined

```
       Leo (o1:Occasion, o2:Occasion) : Boolean == ... ,
```

d1 also has to have the type Occasion. That is to say, the mapping

```
       DateAndTime == Occasion
```

identifies the two types: from the point of view of the lower level, "DateAndTime" is just an alternate name for the very same type also denoted by "Occasion". The mapping causes the upper level typename and the lower level typename to be synonymous.

Our lower level specification will have an extra variable, Reply. In an implementation of the specification, Reply would appear as an output routine. It is curious that a variable can be implemented as a routine. It makes sense in this case only because Reply is a "write-only variable". The effect of an operation never depends on the value that Reply had when the operation was applied.

```
Variable

        Calendar : CalendarType,
        Reply : Message,
        Present : Occasion

Map

        Calendar == Calendar,
        Present == Present
```

RULES FOR MAPPING TYPES, CONSTANTS, AND VARIABLES

There are a few restrictions which mappings must obey in order to make sense. We will sketch them out in order to clarify the way mappings work.

A type mapping associates a type of the higher level with a type at the lower level. The lower-level type must be declared. The mapping causes the upper-level type to be identified with the lower-level type. For instance, an equation between a variable ranging over the higher-level type and a variable ranging over the lower-level makes sense. The upper-level type is just a sketchy description of the lower-level type.

The two predefined types Boolean and Integer are never mapped, because the Ina Jo language processor always maps them to themselves. At the other extreme, a completely unspecified type can be mapped to any type at the lower level. In between the two extremes, there are a few simple, intuitive rules.

An enumerated type may be mapped to another enumerated type with the same number of elements, or else to a set containing that number of successive integers.

If two upper-level types are synonymous, they should be mapped to synonymous lower-level types, or to the same type. If a type is constructed from some base types using "Set of", "><" (Cartesian product), or the other devices for building up complex types, then its image should be constructed from the images of the base types in the same way.

If one upper-level type is a subtype of second, its image under the mapping should be a subtype of the image of the second. If the subtype is defined using the T" operator -- say, as T"x:BaseType (P(x)), then its image should contain just those objects in the image of the base type which satisfy the translation of P. This last condition depends on how the constants of the upper level are mapped to the lower level.

The mappings of variables and constants work in similar ways. The mapping of a variable or constant must match the mappings of the types. A typical variable or constant has parameters, possibly of various types, and yields values of some type. Its declaration therefore looks like:

```
sam(type1, ..., typen) : type0
```

Suppose that the upper-level types type0, type1, ..., typen are mapped to the lower-level types lt0, lt1, ..., ltn. Then sam should be mapped to some expression in the vocabulary of the lower level which, when applied to objects of types lt1, ..., ltn, has a value of type lt0. Ina Jo expresses this in the form:

```
sam(x1, ..., xn) == expr,
```

where expr is some expression containing x1, ..., xn. Expr must make sense when the variables are construed as ranging over types lt1, ..., ltn respectively, and have a value in lt0. There is only one difference between variable mappings and constant mappings. In a constant mapping, expr can contain only constants of the lower level; state variables are not allowed. In a variable map, state variables of the lower level can also appear. The N" operator cannot occur in either case, nor can a transform of the lower level.

For instance, suppose the upper level of a specification, like our Bell-LaPadula file system, has a variable:

```
AccessMatrix(Subject, FileId, Accesskind) : Boolean.
```

Suppose also that we are mapping the top level to a lower level in which there is an access control list associated with each file descriptor. The access control list keeps track of which users are

currently allowed different kinds of access to the file. Thus, at the lower level, we have:

```
Type  . . .
      ACL_Entry = Subject >< Accesskind

Variable
      . . .
      ACL(FileId) : Set of ACL_Entry
```

We would want to map AccessMatrix down to the lower level by stipulating:

```
AccessMatrix(s, f, a) == (s, a) <: ACL(f).
```

CONTINUING OUR EXAMPLE: MAPPING TRANSFORMS

An upper-level transform, say Do_Something(x, y), is mapped using a stipulation of the form:

```
Do_Something(x, y) == expr,
```

where expr may contain x and y, as well as state variables of the lower level. The N" operator is allowed to occur in expr, as are the names of transforms defined on the lower level. As we shall see, a lower-level transform name occurring in expr stands for the conjunction of its refcond and its effects section (augmented by no-change statements covering unmentioned variables).

With the new Reply variable in our example, we can write a much more detailed specification. In the top level version, there was no way to indicate that, say, an attempted addition to the calendar had failed because one or more of the participants were already busy. Thus, we gave the transform Adjoin a refcond stating that all of the participants were free. With the Reply variable we can rewrite the transform to act conditionally. If all of the participants are free, the appointment is added and confirmed in a Reply, while if some of the participants are busy the appointment is not added and the user is warned in the reply. This is an example of our strategy of replacing refconds by conditionals whenever possible.

Note that the next three transforms are not marked External. This means that they are internal transforms. Oddly enough, an internal transform should not be thought of, all on its own, as describing an operation in an abstract machine. Rather, the internal transforms give us a convenient way of expressing the

mappings which carry upper-level transforms down into the vocabulary of the lower level. The "real" transforms -- those which describe the operations of an abstract machine -- are not the internal transforms, but the higher-level transforms mapped down in terms of the internal transforms. Or at least this is so assuming that the upper-level transforms are external, rather than being internal transforms used in mapping down some still higher level of the specification.

```
Transform Update(Now : Occasion)

   Refcond Leo(Present, Now)
   Effect  N"Present = Now
           & N"Calendar = S"a:Appointment
                 (a <: Calendar & Leo(Now, a.When) )

        /*******************************/

Transform AddAppointment( a : Appointment)

   Effect (A" p : Person (p <: a.Who => ~Busy(p, a.When ) )
          =>
              N"Calendar = Calendar || S"( a )
            & N"Reply  = L"( WriteAppointment (a), OK)
          <>
              N"Calendar = Calendar
            & N"Reply  = L"( impossible ) )

        /*******************************/

Transform CancelAppointment( a : Appointment)

   Effect ( a <: Calendar
        =>
          (N"Calendar = Calendar ~~ S"( a )
          &
           N"Reply = L"(WriteAppointment (a), cancelled) )
        <>
          (NC"(Calendar)
          &
           N"Reply = L"(WriteAppointment (a), nonexistent) )
           )
```

We can also add transforms for other useful operations to the lower level. For instance, the calendar will be more useful if it is possible to read off one's appointments for the day.

```
Transform FindAppts (p : Person, d : Date)      External

   Effect A"a:Appointment
               (  (p <: a.Who & d = a.When.WhatDay)
               <->
                  E"n:integer
                       (WriteAppointment (a) = N"Reply.n)
                 )
```

Given a group, we may want to find the earliest time when all the members are free. We may also want to allow an individual to cancel his participation in a meeting.

```
Transform FindTime( g : Group )          External

   Effect  E"o:Occasion
               (N"Reply = L"(WriteTime(o.WhatTime),
                             WriteDate(o.WhatDay))
               &
                A"p: Person (p <: g => ~Busy(p, o) )
               &
                A"o2:Occasion
                ( A"p: Person (p <: g => ~Busy(p, o2) )
                =>
                  Leo(o, o2)  )
               )

Transform Disengage (p: Person, a: Appointment) External

   Effect
        (p <: a.Who
        =>
           N"Calendar = (Calendar ~~ S"(a) )
                        || S"( (a.Who ~~ S"(p), a.When ) )
         & N"Reply = L"(WriteName(p),
                disengaged, WriteAppointment(a) )
        <>
           NC"(Calendar)
         & N"Reply = L"(WriteName(p),
                uninvited, WriteAppointment(a) ) )
```

More importantly, we must say how to interpret the upper-level transforms in the vocabulary of the lower level. For this we use the internal transforms.

```
Map
        UpdatePresent(Now) == Update(Now),
        Adjoin(a) == A"p:Person
```

```
            (p <: a.Who => ~Busy(p, a.When) )
          & AddAppointment (a),
    Remove(a) == a <: Calendar  &
          CancelAppointment (a)
END SecondLevel
END CalendarKeeper
```

What do these mappings mean? An Ina Jo mapping associates an upper-level transform with an expression on the lower level. The lower-level expression must ensure both that the refcond of the upper-level transform is true and that its effect is accomplished.

For instance, since the refcond of Adjoin is

```
A"p:Person (p <: a.Who => ~Busy(p, a.When) ),
```

the first clause in the mapping expression for Adjoin ensures that the refcond is true. When we translate the refcond into the terminology of the lower level, we get the clause itself.

But how does the mapping expression ensure that the effect of Adjoin is accomplished? Note that AddAppointment must not be understood as an instruction: Ina Jo is a descriptive language rather than a procedural language. It has no constructs expressing instructions. Rather, AddAppointment serves as an abbreviation for the conjunction of its refcond (which is vacuously true) and its effects section, supplemented by no-change assertions for variables not mentioned. Hence, the mapping expression is entirely equivalent to:

```
  A"p:Person (p <: a.Who => ~Busy(p, a.When) )
& true
& (A" p : Person (p <: a.Who
                => ~Busy(p, (a.When.WhatTime,
                        a.When.WhatDay) ) )
  =>
      N"Calendar = Calendar
                || S"( (a.Who,
                (a.When.WhatTime, a.When.WhatDay)) )
    & N"Reply  = L"( WriteAppointment
        (a.Who, (a.When.WhatTime, a.When.WhatDay)), OK)
  <>
      N"Calendar = Calendar
    & N"Reply  = L"( impossible ) )

& NC"(Present)
```

Using propositional logic and the type declarations for Occasion and Appointment, we can simplify this to:

```
    A"p:Person (p <: a.Who =>
                          ~Busy(p, a.When) )

  & N"Calendar =  Calendar || S"( a )

  & N"Reply   = L"( WriteAppointment (a), OK)

  & NC"(Present)
```

The Effects section of Adjoin, which is:

```
  N"Calendar = Calendar || S"(a),
```

with NC"(Present) implicit, is thus a consequence of the mapping expression in the most straightforward way. The mapping for the variable Calendar turns the Effects section into a logical consequence of the lower-level expression.

To repeat the point: a mapping expression in Ina Jo has no procedural meaning at all. All it has to do is to imply the refcond and the effects section of the transform being mapped.

THE MEANING OF MAPPINGS

Because the links between levels are logical rather than procedural, the Ina Jo level mechanism has a rather distinctive flavor. Ina Jo levels are not necessarily what one would expect, being, for instance, very different from the levels in the HDM paradigm. In HDM, implementation forges the link between operations in the upper and lower levels of a specification. A program in a standard programming language calling lower-level transforms -- HDM calls them OFUNs -- furnishes a "definition" of an upper-level OFUN in lower-level terms. The mapping gives a procedure which carries out the upper level operation by calling various lower-level operations as subroutines.

The uninitiated sometimes expect Ina Jo levels to work the same way. But it should now be clear that they are quite different. An Ina Jo mapping does not implement the upper-level transform at all. It gives us an expression, couched in the terminology of the lower level. If we manage -- by hook or by crook -- to make this expression true, we will have made the refcond and effect section of the upper-level transform true.

This is a purely descriptive or truth-conditional notion of mapping. The mapping provides us with a way of translating the logical descriptions of the upper-level transforms into the language of the lower level. The result of the translation is a logical formula, couched in lower-level terms. Any state transition caused by the upper-level transform must satisfy the formula.

The FDM notion of mapping has an advantage over the HDM inter-level structure, but it also has a disadvantage. The advantage is that it makes it simpler to define when a mapping is correct. This enabled SDC to write software, included in the Ina Jo language processor, which generates target theorems to test the correctness of mappings. The disadvantage is that the non-procedural approach to mappings is less useful.

The non-procedural approach means that the successive levels of a specification, rather than representing a progression from an initial statement of requirements toward an implementation, are increasingly detailed descriptions at the same logical level. Thus in the process of refining a specification, one does not introduce strategies for transforming the specification into a program. One postpones all issues concerning the structuring of the program itself until the last stage, when one must leap the gulf between the specification, which is purely non-procedural, and the runnable program. Research done by the MITRE Corporation and Odyssey Research Associates [Project 4030, 1985; Platek, 1985] suggests that this issue makes the Ina Jo approach less useful than its alternatives.

The advantage of the Ina Jo inter-level structure is that it is much easier to provide a semantical interpretation of the mappings. In the remainder of this section, I will sketch out the semantics of Ina Jo mappings, discussing the theorems needed to guarantee the correctness of a mapping.

The discussion can be divided into two parts, because one can introduce two different kinds of change in passing from one level to another. First, one can increase the degree of detail in the description of transforms already present at the upper level. This may involve specifying previously unspecified types, adding constants, introducing more concrete data representations for the state variables, and refining the effects sections of transforms. In order to refine the effects sections of upper level transforms, we may want to introduce new internal transforms. The correctness of a mapping which introduces only these sorts of change consists in its fidelity to the decisions which have already been specified at the upper level. The correctness conditions -- criterion and constraint -- specified at the top level will be satisfied automatically at the lower level, assuming only that they were satisfied at the upper

level and that the mappings are faithful to the static and dynamic parts of the upper-level specification. Let us call a mapping a refinement if it introduces only this sort of change.

One can also -- second -- introduce altogether new transforms, by which I mean new external transforms, describing real operations of an abstract machine. These transforms specify operations which did not exist in the machines satisfying the upper-level specification. If we map an upper level to a lower level which has newly added transforms, then we must check that the additional transforms obey the criterion and constraint formulated at the upper level. This is quite different from the checks assuring correctness of the refined detail a mapping may introduce. It corresponds, instead, to the theorems we prove to assure single-level correctness. Let us speak of an extension when we add new external transforms.

It is legitimate to separate out these two elements in the Ina Jo paradigm for levels. For, we can split each mapping into a pure refinement followed by a pure extension. They are independent parts of the mapping process. To split apart the two halves, we simply group together any new external transforms that may appear in the lower level. The remaining part of the lower level amounts to a pure refinement; adding in the external transforms is a pure extension.

Now, the conditions for the correctness of an extension are familiar. Namely, we must prove that the newly introduced transforms preserve the criterion and the constraint from the upper level. Thus, for each newly introduced external transform T, we want to prove:

Map(Criterion) & T => Map(N"Criterion)

and

Map(Criterion) & T => Map(Constraint).

I am using the notation "Map(expr)" to indicate the expression in the lower level vocabulary which we get from expr by applying the translation given in the mapping. We could also add a lower-level invariant to these theorems to get a slighly more general form, closer to the form given in the preceding chapter. Note that it then becomes necessary to prove that the lower-level invariant is at least as strong as the mapped version of the upper-level invariant.

THE FIDELITY OF REFINEMENTS

Simply put, a refinement is faithful if every implementation of the lower level of the specification is an implementation of the upper level. When this is true, we can infer that the lower level specifies, possibly in more stringent detail, the same kind of system the upper level did. In particular, proofs we carried out to ensure that the transforms of the upper level preserve the criterion and constraint will carry over to the lower level. If any machine implementing the lower level could enter a state falsifying the criterion or constraint, then that same machine would implement the upper level and falsify the correctness conditions. The theorems ensure that this cannot happen.

In fact, however, there is a certain amount of purely logical work to be done to work out rigorously this underlying idea. It is not quite the same machine which satisfies the two levels. Rather, given any machine satisfying the lower level, we can construct a very similar machine satisfying the upper level. Let us see what this means, using our distinction between the static and dynamic parts of a specification.

If a mapping is to be faithful, then any structure satisfying the lower level should almost satisfy the upper level. The only hitches we should permit are the following. First, the lower level may have new types in addition to the ones which serve as targets for upper-level types under the mapping. The structure will have sorts of object to interpret these additional types. We may ignore these extraneous sorts when we consider the structure as an interpretation of the upper level. Second, individual constants and function constants belonging to the upper-level may be mapped to complex expressions in the lower level. In this case, we should think of the lower-level structure as equipped with the distinguished objects and functions which these complex expressions define. For instance, in our Calendar example, the ordering relation on the upper level was mapped to a complex expression on the lower level,

```
Lt(t1, t2) == Leo(t1, t2) & t1 ~= t2,
```

where Leo itself had a complex definition:

```
Leo (o1:Occasion, o2:Occasion) : Boolean ==
   (o1.WhatDay = o2.WhatDay
    & Let(o1.WhatTime , o2.WhatTime) )
 | (o1.WhatDay ~= o2.WhatDay
    & Led(o1.WhatDay,  o2.WhatDay) ).
```

Naturally, if a sort of object interprets the lower level type tl, and our mapping carries the upper-level type tu to tl, then the sort must interpret tu. These remarks indicate in what sense the structure for the lower level is "almost" a structure for the upper level.

Now what theorems about the mapping will ensure that every lower-level structure is almost an upper-level structure? Or put the other way round, what problems could prevent the lower-level structure from being almost an upper-level structure?

Assuming, of course, that the mapping satisfies the type compatibility considerations mentioned in Section 3, only the axioms can make problems. We have to make sure that the axioms of the upper level will turn out true in every structure satisfying the lower level. To do so, we must rewrite the upper-level axioms in the vocabulary of the lower level, using the mappings. Moreover, we must prove that the axioms on the lower level entail that the (rewritten versions of the) upper-level axioms are true.

Unfortunately, the Ina Jo language processor does not generate these theorems. This is one of the underlying flaws in the current implementation of the Ina Jo inter-level mechanism. Therefore, any user who relies seriously on multi-level specifications in FDM should make sure that he formulates these theorems by hand, and proves them himself. Without these theorems, there is no assurance whatever that a multi-level design is correct.

Luckily, the Ina Jo processor does better with the dynamic part of a specification; here it at least generates the correct theorems. Let us see what the correct theorems are, by going back to semantics.

What sort of a machine will satisfy the lower level of a pure refinement? The criteria for its underlying structure are just the same as for any other machine: the underlying structure must satisfy the static part of the level. Similarly, the state variables of the machine, and the initial state, relate to the variables and initial condition of the specification in the usual way.

However, since the lower level of a pure refinement has no external transforms, we must say something about the operations of the machine. It should have an operation for each external transform from a higher level which gets mapped down. They contain the state changes that are envisaged by the specifier. Yet, the operations in a machine which satisfies the lower level must satisfy the conditions stated in the mapping expression. Thus in our

Calendar Keeper, we want a machine satisfying the lower level to have an operation corresponding to the Adjoin transform of the upper level. But since the mapping is

```
Adjoin(a) == A"p:Person
        (p <: a.Who => ~Busy(p, a.When) )
        & AddAppointment (a.Who,
                a.When.WhatTime, a.When.WhatDay )
```

This, in turn, expanded and simplified, became:

```
           A"p:Person (p <: a.Who =>
                              ~Busy(p, a.When) )

        &  N"Calendar =
                   Calendar || S"( a )

        & N"Reply  = L"( WriteAppointment (a), OK)

        & NC"(Present)
```

This is the specification that the operation should fulfil.

This machine may have state variables which are mentioned only at the lower level, not at the upper level. To transmute the machine for the lower level into a machine for the upper level, we may have to "throw away" these extra state variables. That is, we define the operations of the new machine so that their result does not depend on the values of the additional state variables.

Let us say that a state s, involving only the variables of the new machine, contracts s′, involving all the variables of the lower-level machine, if s and s′ agree on all the variables which are in the new machine. Then we can define the (generally non-deterministic) operation O of the new machine, corresponding to an operation O′ belonging to the lower-level machine, as follows.

The state t is a possible next state for the machine, when O is applied in state s with parameters p1 ... pn, just in case there are states s′ and t′, such that s′ and t′ contract to s and t respectively, and t′ is a possible next state when O′ is applied to p1 ... pn in state s′. If M′ is a machine for the lower level, then the machine we get by applying this definition to all operations O′ in M′ can be called M.

The form of this definition means that M will often be nondeterministic even when M′ is deterministic: the choice of different values of s′ may lead to different possible next states.

Were it not for this fact, it would be possible to carry out the semantics of Ina Jo in terms of deterministic machines exclusively.

What theorems about the mapping will ensure that M actually satisfies the upper level of the specification, assuming that M′ satisfies the lower level? We must prove, for each transform, that its lower-level specification entails its upper-level specification, as translated into the vocabulary of the lower level by the mapping. We must prove:

LowerLevelSpec => Map(Refcond) & Map(Effects).

The Ina Jo language processor generates a theorem of this form for each external transform from a higher level.

Of course there is one other fact which we must also prove: namely, that the initial state of M satisfies the initial condition of the upper level. The initial state of M is just the result of contracting the initial state of M′ to the state variables which actually occur in M. To establish that the initial state of M satisfies the initial condition at the higher level, it is enough *show that the initial state of M′ satisfies it.* All we know about the initial state of M′ is that it satisfies the initial condition of the lower level. The theorem we want, then, is:

LowerLevelInit => Map(UpperLevelInit).

These theorems, then, ensure the correctness of a refinement.

CONCLUSIONS

With these remarks on the correctness of mappings we complete an extensive analysis of the design of the Ina Jo language. We have tried to provide examples to illustrate the workings of all of the major features of the language. Moreover, we have tried to point out flaws in its design. These include the leveling mechanism, the Refcond, and the type abstraction operator T". We have also pointed out a number of points at which the logic embodied in the language or language processor suffers from gaps or actual inconsistencies. Moreover, we have argued that the conception of correctness embodied in Ina Jo′s Criterion and Constraint is not completely general, and that it does not allow us to express certain important kinds of requirement. Nevertheless, the Ina Jo specification language is lucid and flexible. It is a valuable tool on the specifier′s workbench.

REFERENCES

1. Digicomp Research Corporation, "Verification Methodology Evaluation (VMAN)," forthcoming.

2. Dijkstra, E. W., Structured Programming, New York: Academic Press, 1972.

3. Eggert, P. R. "Overview of the Ina Jo specification Language," SDC TR-SP 4082, October, 1980.

4. Goguen, J. A. and J. Meseguer, "Security Policies and Security Models," Proc. 1982 Symp. Security and Privacy, IEEE No. 82CH1753-3, 11-20.

5. Goguen, J. A. and J. Meseguer, "Unwinding and Inference Control," Proc. 1984 Symp. Security and Privacy, IEEE No. 84CH2013-1, 75-86.

6. Huff, G. A., "A Pre-analysis of the KVM Formal Specifications," WP 23993, The MITRE Corporation, Bedford, MA, 1981.

7. Kramer, S. M., "The Ina Jo Flow Table Generator," WP 23103, The MITRE Corporation, Bedford, MA, 1981.

8. Kramer, S. M., "Information Flow Security Analysis for KVM/370 Specifications," MTR 8799, The MITRE Corporation, Bedford, MA, 1982.

9. Korelsky, T. and D. Sutherland, "Formal Specification of a Multi-Level Secure Operating System," Proc. 1984 Symp. Security and Privacy, IEEE No. 84CH2013-1, 209-18.

10. Millen, J. K. and C. M. Cerniglia, "Computer Security Models," MTR 9531, The MITRE Corporation, Bedford, MA, 1984.

11. Locasso, R., J. Scheid, V. Schorre, P. Eggert. "The Ina Jo Specification Language Reference Manual," SDC TM-(L)-6021/001/00, June 1980.

12. Project 4030 Staff, "An Update Review of the Ina Jo Verification Condition Generator," The MITRE Corporation Working Paper 25972, Bedford, MA, 1985.

REFERENCES (Concluded)

13. Platek, R., "An Update Review of the Ina Jo Implementation Level," The MITRE Corporation Working Paper 25971, Bedford, MA, 1985.

14. Platek, R. and D. Sutherland, "The Semantics of the Feiertag MLS Information Flow Tool and Its Impact on Design Verification: Some SCOMP Examples," Odyssey Research Associates, 1984.

15. Scheid, J., J. Landauer "Restaurant: An example of the Ina Jo Software Development Methodolgy," SDC TM-7043/000/00, October 1980.

16. Scheid, J.,"The Design of the Ina Jo Verification Condition Generator (VCG) for Modula," SDC TM-7393/000/00, September, 1983.

17. Scheid, J., "Modula VCG User Manual," SDC TM-7393/001/00, September, 1983.

18. Scheid, J., "Implementation Specification," SDC TM-7315/000/00, September 1983.

19. Scheid, J., "Conversion of FDM to Multics: Software Requirements Specification for Enhancements," SDC TM-7413-005/01, February, 1984.

20. Schorre, V., J. Stein " The ITP User Manual," SDC TM-6889/000/03, August 1983.

21. Stein, J., D. Gillmann "How to Prove Transform Theorems Using the ITP Version 11," SDC, TM-(L)-6021/003/00, Dec. 1980.

22. Tanenbaum, A. S., Structured Computer Organization, 2nd ed., Englewood Cliffs: Prentice-Hall, 1984.

DISTRIBUTION

INTERNAL

D-10

A.J. Tachmindji

D-70

W.S. Attridge
E.H. Bensley
E.L. Lafferty
R.J. Sylvester

D-75

L.L. Abraham
F. Belvin
T.C. Vickers Benzel (5)
J.L. Berger
D.J. Bodeau
M.J. Brooks
R.K. Burns
C.M. Cerniglia
F.N. Chase
M.H. Cheheyl
S.C. Clark
B.E. Davis
M. Daya
K.J. Duffy
J.W. Francis
J.J. Glass
T.P. Gleason
H.G. Goldman
J.D. Guttman (10)
H.L. Hall
K.L. Hasler
R.T. Jordan
D.N. Juitt
J.K. Millen
S.L. Miravalle
J.F. Myers
L.R. Scott
R.M. Smaby
J. Sullivan
P.S. Tasker
D.A. Tavilla
M.M. Zuk

INTERNAL (continued)

D-78

T.C. Antognini
C.H. Applebaum
E.H. Bensley
S.H. Brackin
D.L. Drake
W.M. Farmer
J.L. Katz
L.G. Monk
R.D. Silverman
P.J. Simpson
F.J. Thayer
R.J. Watro
J.G. Williams

EXTERNAL

John C. Faust, RADC/COTC (5)
Rome Air Development Center
Griffiss Air Force Base
Rome, NY 13441

Capt. J. Itz, ESD/ALSE (5)
Electronic Systems Division
Hanscom Air Force Base
Bedford, MA 01731

END

12-86

DTIC